R Supplement to "Mathematics for the Life Sciences"

Erin N. Bodine, Suzanne Lenhart & Louis J. Gross
Supplement R codes developed by Tyler Massaro

1.3 R Skills

If you are not familiar with the software R, please review "Getting Started with R" in the Appendix.

Entering Data Sets in R

In R data sets are entered as arrays, and arrays are denoted with the concatenate function: `c()`. If we wanted to enter the trees per hectare data from Example 1.2, we would type

```
c(13.3, 13.5, 24.6, 18.7, 10.9)
```

into R. Notice that each data point in the set is separated by a comma. If we want to refer back to this data set using R, we need to name in the data set. In Example 1.2 we called the data set x. To call the data x in R, we type

```
x = c(13.3, 13.5, 24.6, 18.7, 10.9)
```

into R. Now, whenever we want to refer back to our data set we can just use x instead of typing the entire data set again.

Calculating Descriptive Statistics in R

Now that we know how to enter our data sets into R, we would like to use R to quickly compute basic descriptive statistics. Table 1.1 shows the commands for the descriptive statistics described earlier in this chapter.

Command	Description
`mean(x)`	returns aritmetic mean of data set x
`geometric.mean(x)`†	returns geometric mean of data set x
`harmonic.mean(x)`†	returns harmonic mean of data set x
`median(x)`	returns median of data set x
`mfv(x)`‡	returns mode of data set x (when there are multiple values occurring equally frequently, `mfv(x)` returns all of these values)
`min(x)`	returns minimum value of data set x
`max(x)`	returns maximum value of data set x
`var(x)`	returns the variance of data set x
`sd(x)`	returns the standard deviation of data set x

Table 1.1: R commands for a variety of descriptive statistics. In each case, x refers to the data set.
† *requires the* psych *package.* ‡ *requires the* modeest *package.*

Each of the commands in Table 1.1 returns its corresponding answer. If we wish to save the answer for future use, we must name the output of the command. For example, if we wish to save the aritmetic mean we can type

```
xbar = mean(x)
```

into R. If you are typing this into the command window, you may check to make sure the name was assigned correctly by typing `xbar` into the command line and pressing enter. The value returned will be your desired arithmetic mean.

Notice there are no commands for calculating the range or the midrange. We can calculate these, however, by using the min and max commands. To calculate the midrange we use

```
(min(x)+max(x))/2
```

and to calculate the range we use

```
max(x)-min(x)
```

As an example, suppose we wanted to calculate the mean, median, mode, midrange, geometric mean, harmonic mean, range, variance, and standard deviation for the data set in Example 1.1.

The following shows in the input type into the command window (always proceeded by `>`), and its corresponding output.

Command Window

```
> require(psych)
> require(modeest)
>
> y = c(65, 70, 90, 95, 82, 84, 61, 83, 120, 83, 72, 70,
+ 72, 71, 92, 85, 102, 69)
> y
 [1]  65  70  90  95  82  84  61  83 120  83  72  70  72
+ 71  92  85 102  69
>
> ybar = mean(y)
> ybar
[1] 81.44444
>
> ymed = median(y)
> ymed
[1] 82.5
>
> ymode = mfv(y)
> ymode
[1] 70 72 83
>
> ymidrange = (min(y)+max(y))/2
> ymidrange
[1] 90.5
>
> ygeo = geometric.mean(y)
> ygeo
[1] 80.27469
>
> yharm = harmonic.mean(y)
> yharm
[1] 79.1871
>
> yrange = max(y)-min(y)
> yrange
[1] 59
>
> yvar = var(y)
> yvar
[1] 217.3203
>
> ystd = sd(y)
> ystd
[1] 14.74179
```

2.5 R Skills

Histograms

The command in R used to make histograms is

```
hist(x)
```

where x is the data set. R will automatically execute Sturges' formula to determine an appropriate number of classes for the data. However, the `hist()` command may call a separate argument to alter the number of classes, which we will see later.

In fact, `hist()` may take any number of additional arguments to change various attributes of the histogram. Consider the following line of code below, which creates a histogram with a title and axes:

```
hist(x, main = ‘‘Our Title’’, xlab = ‘‘Ages’’, ylab = ‘‘Number’’)
```

Here, we use the argument `main` to add a title, and `xlab` and `ylab` to add corresponding axis labels.

The R script (see Appendix for description of R scripts) used to create the histogram shown in Example 2.2 is shown below.

BlackBearHist.r

```
> # R-script to create Female Black Bear Histogram
>
> # Enter data
> w = c(60, 85, 95, 85, 115, 75, 140, 145, 120, 110, 90, 115,
+ 75, 125, 80, 80, 80, 110, 75, 120, 150, 38, 118)
>
> # Calculate range
> range = max(w)-min(w)
>
> # Calculate class width
> cw = ceiling(range/6)
>
> # Determine values where bars should start
> startvals = seq(min(w)-0.5,min(w)-0.5+6*cw,by=cw)
>
> # Make Histogram with labels
> hist(w,
+      breaks = startvals,
```

```
+      xlab = "Female Black Bear Weight (lbs)",
+      ylab = "# of Female Black Bears",
+      xlim = c(min(w)-0.5,min(w)-0.5+6*cw),
+      xaxt = "n")
> axis(1,
+      at = startvals)
```

Refer to the appendix for descriptions of the function `ceiling` and how to form an array using the structure `seq(a,b,by=c)`.

In general, one may use multiple lines to make assignments or list arguments. For example, in the above set of commands, the assignment of `w`, and the list of arguments for `hist` and `axis` are all on multiple lines. When R executes these commands, it will replace the `>` with a `+` until it reaches the end of the command.

For future reference, the option `xaxt = "n"` turns off all tick marks and corresponding labels on the x-axis. Similarly, the 1 passed to `axis` refers to what R considers to be the first axis, or, in our case, the x-axis. It then puts tick marks back in, `at` the values we stored in `startvals`.

Scatter Plots

If you have a set of n points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, the command in R to make a scatterplot of the x values versus the y values is

```
plot(x,y)
```

where `x` is the array for the set $x = \{x_1, x_2, \ldots, x_n\}$, and `y` is the array for the set $y = \{y_1, y_2, \ldots, y_n\}$.

Suppose you have the following data

x	2	5	2	4	6
y	4	7	5	8	11

To make a scatter plot you could type the following into the Command Window of R

Command Window

```
> x = c(2, 5, 2, 4, 6)
> y = c(4, 7, 5, 8, 11)
> plot(x,y)
```

When you press "Enter" after the last command, a new window will pop up in R, one containing an image of the scatter plot.

We can add axis labels using the same `xlab` and `ylab` options we used with `hist`. The following sequence of commands typed into the Command Window of R will produce a scatterplot with axis labels.

Command Window

```
> x = c(2, 5, 2, 4, 6)
> y = c(4, 7, 5, 8, 11)
> plot(x,y,
+        xlab = "An x label",
+        ylab = "A y label")
```

The `plot` function also allows us to make plots of lines and curves. Suppose we want to plot the line given by the equation $f(t) = 1.25t + 2.25$ and the curve given by the equation $g(t) = t^3 - 5t^2 + 8t + 3$. Since the `plot` function plots sets of points, it would seem we must first create the $(t, f(t))$ and $(t, g(t))$ sets of points to plot. However, the `plot` function is clever, and if we define the set of t points to use, it can compute the corresponding $f(t)$ and $g(t)$ values within the `plot` function. Suppose we want to graph both functions from $t = 0$ to $t = 8$. Then we would create the t points using the command

```
t = seq(0, 8, by = 0.1)
```

This command creates a vector with 81 entries. The first entry is 0, the second entry is 0.1, and the entries continue to increase by increments of 0.1 until they reach the value of 8.0. Now, we can plot our two functions $f(t)$ and $g(t)$.

Command Window

```
> t = seq(0, 8, by = 0.1)
> plot(t, 1.25*t + 2.25,
+        type = "l")
> plot(t, t^3 - 5*t + 8*t + 3,
+        type = "l")
```

You should quickly notice that R actually removes any and all graphics from its "Quartz" window before executing the second plot command. We can change this by using the `par(new=TRUE)` command (see below):

Command Window

```
> t = seq(0, 8, by = 0.1)
> f = 1.25*t + 2.25
> g = t^3 - 5*t^2 + 8*t + 3
>
```

```
> ymin = min(f,g)
> ymax = max(f,g)
>
> plot(t, f,
+        type = "l",
+        col = "red",
+        main = "",
+        xlab = "t",
+        ylab = "",
+        ylim = c(ymin, ymax))
>
> par(new = TRUE)
>
> plot(t, g,
+        type = "l",
+        col = "green",
+        main = "",
+        xlab = "",
+        ylab = "",
+        xaxt = "n",
+        yaxt = "n")
>
> legend("topright",
+        c("f(t)", "g(t)"),
+        lty = c(1,1),
+        col = c("red","green"),
+        bg = "white")
```

This block of code may require some explanation. We'll go in order:

- `f = 1.25*t + 2.25` allows us to cheat and get f evaluated at each t,
- `ymin` takes the minimum value achieved by $f(t)$ and $g(t)$,
- `type = "l"` connects our points with a line,
- `col = "red"` makes the line red,
- `main = ""` makes a title with no characters (i.e., no title),
- `ylab = ""` makes a y-label with no characters,
- `ylim = c(ymin,ymax)` sets the limits on our y-axis,
- `par(new = TRUE)` tells R to put the next graphic in the same window,
- `xlab = ""` makes an x-label with no characters (since we already have an x-label from the first plot),

- `xaxt = "n"` removes all tick marks and corresponding numbers from the x-axis (again, since we already have those from the first plot),
- `"topright"` tells R where to put the legend,
- `c("f(t)","g(t)")` fills in the labels for the legend,
- `lty = c(1,1)` acts essentially like `type = "l"` (`lty` stands for "line type"),
- `col = c("red","green")` colors the lines in the legend,
- `bg = "white"` makes the background of the legend white (otherwise it will be transparent and you will see the graph behind it).

To find further explanation for all of these options, or the `plot` command in general, type

Command Window

```
> ?plot()
```

into the Command Window.

3.5 R Skills

Linear Regression

R can easily determine the slope and y-intercept for the "best" line through a set of data. Use the command

```
C = lm(Y~X)
```

which produces a list `C` in which the first value is the best fit for the intercept and the second value is the best fit for the slope for the least-squares fit of the vector of data `Y` (on the vertical axis) to the vector of data `X` (on the horizontal axis).

For example, if we wanted to find the least square regression line for the data in Example 3.1, we could type into the Command Window

Command Window

```
> x = c(2, 5, 2, 4, 6)
> y = c(4, 7, 5, 8, 11)
> C = lm(y~x)
> coef(C)
(Intercept)           x
    1.65625     1.40625
```

Thus, the equation for the least-squares regression line would be

$$\hat{y} = 1.40625x + 1.65625.$$

Interpolation and Extrapolation

To interpolate and extrapolate in R use the command

```
yhat = predict(C, data.frame(x = xhat))
```

where `C` is the linear model predicted earlier, `xhat` is the horizontal axis value at which you would like to estimate the corresponding vertical axis value, and `yhat` is the corresponding vertical axis value.

Continuing from our example above, to find estimated y values for $x = 3$ and $x = -1$ using the linear regression for the data in Example 3.1 we could type the following into the Command Window

Command Window

```
> yhat1 = predict(C, data.frame(x = 3))
> yhat1
    1
5.875
> yhat2 = predict(C, data.frame(x = -1))
> yhat2
   1
0.25
```

Correlation Coefficients

Again, R makes it easy to calculate the correlation coefficient of two vectors using

```
rho = cor(X,Y)
```

where `X` is the data on the horizontal axis, `Y` is the data on the vertical axis, and `rho` is the value of the correlation coefficient. Note that the correlation coefficient is a dimensionless number – in calculating it the dimensions of the measurements cancel out.

Continuing from our example above, if we want to find the correlation coefficient for the data in Example 3.1 we could type the following in the Command Window

Command Window

```
> rho = cor(x,y)
> rho
[1] 0.9185587
```

Thus, the correlation coefficient for our data set is $\rho = 0.9186$.

Although in the examples above we typed all of our commands into the Command Window, we could also write an R script that executes all of these commands. Below is an example of such an R script.

LSR.R

```
# Filename: LSR.R
# R script to
#   - compute equation for least squares regression line
#   - plot least squares regression line on a graph with the data
#   - compute the correlation coefficient
```

```
#   - compute the coefficient of determination
# LSR = least square regression

# Enter the data
x = c(2, 5, 2, 4, 6)
y = c(4, 7, 5, 8, 11)

# Find the equation for the LSR line
C = lm(y~x)
# Display the equation
cat(sprintf("Eqn for LSR: yhat = %f x + %f",coef(C)[2],
    coef(C)[1]), "\n")

# Find the yhat value for each x value
yhat = predict(C, data.frame(x))

# Plot the data and the LSR line
plot(x,y,
   pch  = 20,
   xlab = "x",
   ylab = "y",
   xlim = c(min(x)-1, max(x)+1),
   ylim = c(min(y)-1, max(y)+1))
par(new = TRUE)
plot(x, yhat,
   type = "l",
   xlab = "",
   ylab = "",
   xlim = c(min(x)-1, max(x)+1),
   ylim = c(min(y)-1, max(y)+1),
   xaxt = "n",
   yaxt = "n")

# Find the correlation coefficient
rho = cor(x,y)
# Display the correlation coefficient
cat(sprintf("rho = %f", rho), "\n")
# Display the coefficient of determination
cat(sprintf("The regression line accounts for %2.2f%% of
    the variance in the data", 100*rho^2), "\n")
```

See Appendix A for details on creating and running an R script. When this R script is run in the terminal, the output looks like

Command Window

```
> source("LSR.R")
Eqn for LSR: yhat = 1.406250 x + 1.656250
rho = 0.918559
The regression line accounts for 84.38% of
    the variance in the data
```

The graph is shown in Figure 3.2.

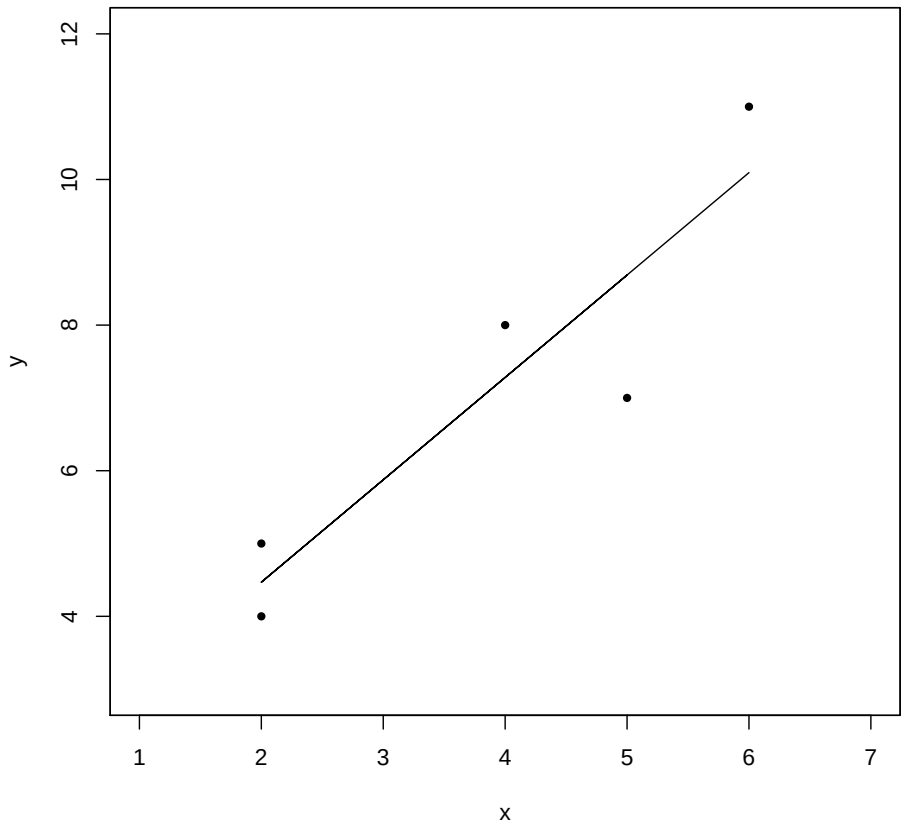


Figure 3.2: Graph produced by the R script LSR.R.

4.5 R Skills

Exponential & Logarithmic Functions in R

In R, if we want to compute the value e^x, where x is some number, we use the function

```
exp(x)
```

This function can also be used on arrays of numbers. If `x` is an array, then `exp(x)` will return an array where each value is e raised to the corresponding value in `x`. Examples of using `exp` on a number and on an array are both shown below.

Command Window

```
> a = 5
> x = c(1, 2, 3, 4, 5)
>
> exp(a)
[1] 148.4132
> round(exp(x), 4)
[1]   2.7183   7.3891  20.0855  54.5982 148.4132
```

The function for computing the natural logarithm in R, `log(x)`, works similar to `exp(x)`. Note that R uses `log` for computing the natural logarithm. If you want to compute the logarithm of a different base, you need to use the formula

$$\log_a x = \frac{\ln x}{\ln a}.$$

For example,

$$\log_{10} 5 = \frac{\ln 5}{\ln 10}.$$

Like the `exp` function, the `log` function can be applied to an array. If `log(y)`, where `y` is an array of numbers, then the function will return an array where each value is the natural logarithm of the corresponding value in the array `y`. Some examples of using the function `log` are shown below.

Command Window

```
> a = 5
> b = 10
> x = c(1, 2, 3, 4, 5)
>
```

```
> log(a)
[1] 1.609438
> log(b)/log(10)
[1] 1
> y = log(x)
> round(y, 4)
[1] 0.0000 0.6931 1.0986 1.3863 1.6094
> z = exp(y)
> z
[1] 1 2 3 4 5
```

Rescaling Data and Linear Regressions

In Section 4.4 we learned how to rescale data so that we could use our linear regression techniques to fit a least-squares regression line to the data. Let us see how we can use R to accomplish this task.

In Example 4.10 we looked at the total pounds of bluefish caught in the Chesapeake Bay every five years. In the example, we rescale the data by letting $x = \frac{1}{5}(\text{year} - 1940)$ (a linear rescaling), and $\ln = \ln(\text{lbs of bluefish})$ (a logarithmic rescaling). Next, we plotted the data. Then we computed the equation for the least-squares regression line for the $(x, \ln y)$ data. Below is an R script that does all these calculations in R. Additionally, the R script computes the correlation coefficient for the $(x, \ln y)$ data.

Bluefish.R

```
# Filename: Bluefish.R
# R script to
#    - enter bluefish data
#    - rescale bluefish data
#    - compute equation for least squares regression line
#    - plot least squares regression line on a graph with the data
#    - compute the correlation coefficient
# LSR = Least Squares Regression

# Enter year array
year = seq(1940, 1990, by = 5)
# Rescale year array to get x data
x = (year - 1940)/5

# Enter pounds of bluefish caught
y = c(15000,
   150000,
   250000,
   275000,
```

```
   270000,
   280000,
   290000,
   650000,
   1200000,
   1500000,
   2750000)

# Find the equation for the LSR line
C = lm(log(y)~x)
# Display the equation
cat(sprintf("Eqn for LSR: ln y = %f x + %f", coef(C)[2],
    coef(C)[1]), "\n")

# Find the lnyhat value for each x value
lnyhat = predict(C, data.frame(x))

# Plot the data and the LSR line
plot(x,log(y),
   pch  = 20,
   xlab = "Year (rescaled)",
   ylab = "ln(Pounds of bluefish caught)",
   xlim = c(min(x)-1, max(x)+1),
   ylim = c(min(log(y))-1, max(log(y))+1))
par(new = TRUE)
plot(x, lnyhat,
   type = "l",
   xlab = "",
   ylab = "",
   xlim = c(min(x)-1, max(x)+1),
   ylim = c(min(log(y))-1, max(log(y))+1),
   xaxt = "n",
   yaxt = "n")

# Find the correlation coefficient
rho = cor(x,log(y))
# Display the correlation coefficient
cat(sprintf("rho = %f", rho), "\n")
```

See Appendix A for details on creating and running an R script. When this R script is run in the Command Window, the output looks like

Command Window

```
> source("Bluefish.R")
Eqn for LSR: ln y = 0.379678 x + 10.878423
```

```
rho = 0.908055
```

The graphical output is shown in Figure 4.1.

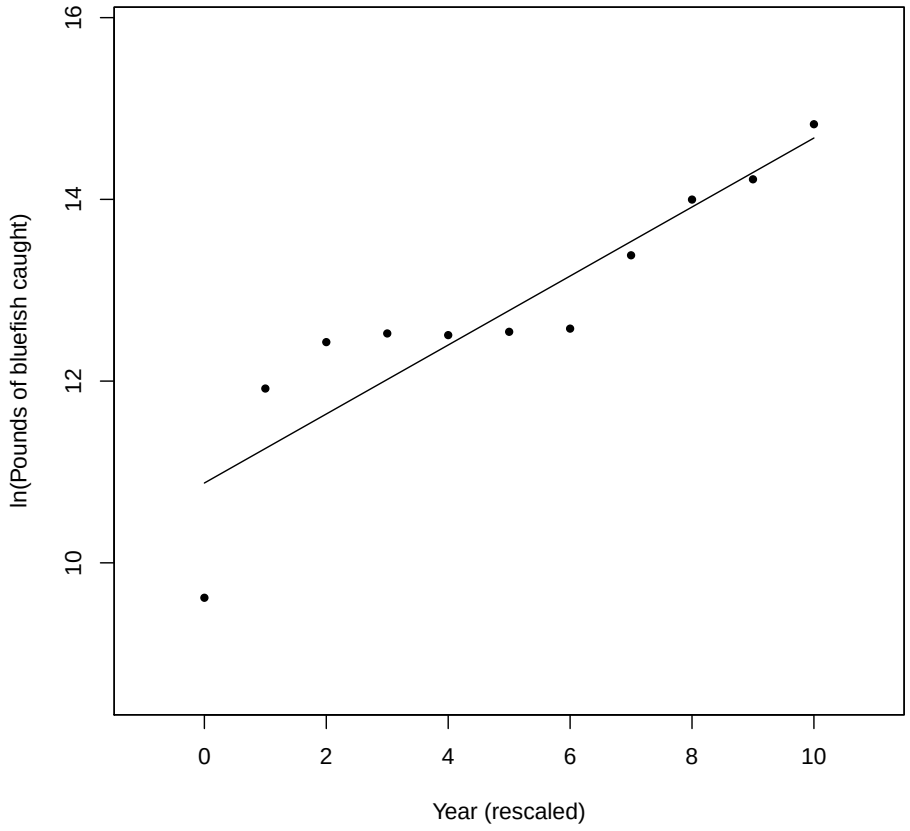


Figure 4.1: Graph produced by the R script Bluefish.R.

Sometimes we are given a set of data and need to decide which type of function (linear, exponential, or allometric) best describes how the data are related (see Exercises 4.12, 4.13, and 4.14). In order to do this, we compare the correlation coefficients of the original data, with (1) the data logarithmically rescaled only in the vertical axis variable, and (2) the data logarithmically rescaled in both the horizontal and vertical axis variables.

Suppose we are given data and must decide what type of function best describes the relationship of the data. The data in Table 4.1 are for the body weights in

grams (g) and pulse rate in beats per minute (bpm) for various mammals. The data are from [14, 25].

The following is an R script that computes the correlation coefficient for the (x, y) data, for the $(x, \ln y)$ data, and for the $(\ln x, \ln y)$ data.

Mammal	Body Weight (g), x	Pulse Rate (bpm), y
Vesperugo pipistrellus	4	660
Mouse	25	670
Rat	200	420
Guinea Pig	300	300
Rabbit	2000	205
Little dog	5000	120
Big dog	30,000	85
Sheep	50,000	70
Man	70,000	72
Horse	450,000	38
Ox	500,000	40
Elephant	3,000,000	48

Table 4.1: Data on mammal pulse rates relative to body weight [14, 25].

MammalPulseRates.R

```
# Filename: MammalPulseRates.R
# R script to
#    - enter mammal pulse rates data
#    - calc. correlation coeff. for y vs x
#    - calc. correlation coeff. for ln y vs x
#    - calc. correlation coeff. for ln y vs ln x

# Enter the data
x = c(4,
   25,
   200,
   300,
   2000,
   5000,
   30000,
   50000,
   70000,
   450000,
   500000,
   3000000) # body weight data
y = c(660,
   670,
   420,
```

```
   300,
   205,
   120,
   85,
   70,
   72,
   38,
   40,
   48) # pulse rate data

# Calculate correlation coeff. for y vs x
rho = cor(x,y)
# Display the correlation coefficient
cat(sprintf("(x, y)          rho = %f", rho), "\n")

# Calculate correlation coeff. for ln y vs x
rho = cor(x, log(y))
# Dsplay the correlation coefficient
cat(sprintf("(x, ln y)       rho = %f", rho), "\n")

# Calculate correlation coeff. for ln y vs ln x
rho = cor(log(x), log(y))
# Display the correlation coefficient
cat(sprintf("(ln x, ln y)    rho = %f", rho), "\n")
```

When this R script is run in the Command Window, the output looks like

Command Window

```
> source("MammalPulseRates.R")
(x, y)          rho = -0.334064
(x, ln y)       rho = -0.442814
(ln x, ln y)    rho = -0.976793
```

From the output, we see that when both the x and y data are scaled logarithmically, the correlation coefficient has the largest absolute value. Thus, if choosing between a linear, exponential, and allometric function, an allometric function will best fit the data. To find the equation of the allometric function, we use the `lm` function to obtain the equation of the line (where ln x is the horizontal variable and ln y is the vertical variable), and then transform the function into an equation of y in terms of x.

Command Window

```
> lm(log(y)~log(x))

Call:
```

```
lm(formula = log(y) ~ log(x))

Coefficients:
(Intercept)       log(x)
     7.0372      -0.2461
```

Thus, the equation for the least-squares regression line is

$$\ln y = -0.2461 \ln x + 7.0372.$$

We can then solve this equation for y in terms of x.

$$\begin{aligned}
\ln y &= -0.2461 \ln x + 7.0732 \\
e^{\ln y} &= e^{-0.2461 \ln x + 7.0732} \\
y &= e^{\ln x^{-0.2461}} \cdot e^{7.0732} \\
&= (x^{-0.2461})(1179.9) \\
&= 1179.9 x^{-0.2461}.
\end{aligned}$$

5.7 R Skills

Given a difference equation $x_{n+1} = f(x_n)$, we would like to utilize R to generate a plot of x_n for some finite set of n values. To simulate a discrete difference equation model, we first need to learn how to implement *for loops* in R.

Loops

Often we would like to use R to perform the same operation over and over with only a slight change. It is tedious to type the same commands (with a slight change) over and over. To remove the tediousness of this task, we use what are known in computer programming as ***for loops***. The basic structure of a for loop is

```
for (i in # some array of values){
   # some commands that change only as i changes
   }
```

where `i` is known as the ***index*** of the loop. Though it is typical to use the letter `i` for the index, you may use whatever label you like. Other common index labels are `j`, `k`, and `count`.

Suppose we wanted to sum up the numbers from 1 to 100. We could do this with for loop.

Sum100.R

```
# Filename: Sum100.R
# R script to
# - Sum the numbers 1 to 100

total = 0                # initial total to 0
for (i in 1:100){        # loop through 1, 2, ..., 100
   total = total + i     # add next value to total
   }
cat(sprintf("total = %i", total), "\n")
```

When this R script is run in the Command Window, the output looks like

Command Window

```
> source("Sum100.R")
total = 5050
```

Suppose that we want to model a population that grows according to the difference equation

$$x_{n+1} = \underbrace{x_n}_{\text{Population density at time step } n} + \underbrace{rx_n(1-x_n)}_{\text{Growth term}},$$

where the value r is referred to as the *instrinsic growth rate* of the population. This difference equation is known as the ***logistic difference equation***, and the general solution of this equation cannot be found using the methods presented here. Thus, we will use R to explore what happens to the population over some finite number of n values. As with `Sum100.R` above, we use a ***for loop*** to determine the values of x_n. We will use an instrinsic growth rate of 0.8 and an initial value of $x_0 = 0.2$:

`LogisticDifferenceEqn.R`

```
# Filename: LogisticDifferenceEqn
# R script to simulate the logistic difference equation

# Set the values of r and x0
r = 0.8
x0 = 0.2

# Through the loop we will fill the values x_n into the array x.
# In the first iteration of the loop, the value of x[1] needs to
# be known.  We set that value before starting the loop
x = numeric()
x[1] = x0

for (n in 1:50) {
   x[n+1] = x[n] + r*x[n]*(1-x[n])
   }

# Plot the results
plot(x,
     type = "b",
     pch = c(20),
     xlab = "Time Step",
     ylab = "Population  Density",
     main = "")
```

When we run this file, we generate the graph shown in Figure 5.4.

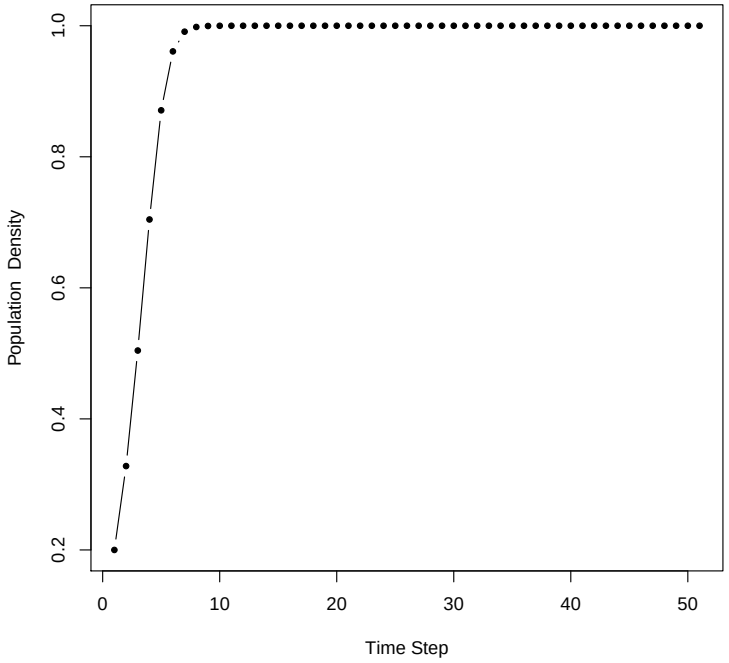


Figure 5.4: Output from LogisticDifferenceEqn.R.

6.4 R Skills

See Appendix A.3 for information on how to work with vectors and matrices in R.

7.3 R Skills

Matrix Operations in R

Since R is built to easily handle matrices, basic matrix operations in R are what you would expect. If you have two matrices of the same size and want to add or subtract them, use the `+` operator or `-` operator, respectively. If you have two matrices that you want to multiply (and each is the appropriate size), then use the `%*%` operator. If you try to add or subtract matrices that do not have the same dimension, or try to multiply matrices that do not have compatible dimensions, R will return an error message. R also makes it easy to multiply a scalar number by a matrix. Now, we use the `*` operator.

Command Window

```
> A = matrix(c(5, 2, 1, 3), ncol = 2)
> A
     [,1] [,2]
[1,]    5    1
[2,]    2    3
> B = matrix(c(3, 1, -1, 0, 0, 2), ncol = 3)
> B
     [,1] [,2] [,3]
[1,]    3   -1    0
[2,]    1    0    2
> C = matrix(c(-1, 7, 5, 0), ncol = 2)
> C
     [,1] [,2]
[1,]   -1    5
[2,]    7    0
> A + C
     [,1] [,2]
[1,]    4    6
[2,]    9    3
> A - C
     [,1] [,2]
[1,]    6   -4
[2,]   -5    3
> A%*%B
     [,1] [,2] [,3]
[1,]   16   -5    2
[2,]    9   -2    6
> A%*%C
     [,1] [,2]
[1,]    2   25
[2,]   19   10
```

```
> 5*A
     [,1] [,2]
[1,]   25    5
[2,]   10   15
> (1/5)*A
     [,1] [,2]
[1,]  1.0  0.2
[2,]  0.4  0.6
> A/5
     [,1] [,2]
[1,]  1.0  0.2
[2,]  0.4  0.6
> B%*%A
Error in B %*% A : non-conformable arguments
> A-B
Error in A - B : non-conformable arrays
> B+A
Error in B + A : non-conformable arrays
```

Notice the last three commands produced error messages. Notice also that `(1/5)*A` produces the same matrix as `A/5`. Thus, we see that we can use the `/` operator as a shortcut for multiplying by $\frac{1}{5}$.

In addition to having these standard operations, R has a few operations that make working with matrices easier. Suppose we want to subtract a constant value from every entry in a matrix. For example, suppose we wanted to subtract 2 from every entry in

$$\mathbf{A} = \begin{bmatrix} 5 & 1 \\ 2 & 3 \end{bmatrix}.$$

To do this by hand, we would calculate

$$\begin{bmatrix} 5 & 1 \\ 2 & 3 \end{bmatrix} - 2 \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

However, R has a short hand for this. In R, we can type `A-1` to get the same result. When we subtract a scalar value from a matrix in R, R assumes we want to multiply that scalar number by the appropriate size matrix filled with 1's.

Command Window

```
> A = matrix(c(5, 2, 1, 3), ncol = 2)
>
> A - 2
     [,1] [,2]
[1,]    3   -1
[2,]    0    1
```

```
> A + 2
     [,1] [,2]
[1,]    7    3
[2,]    4    5
```

Creating Tables of Output

We can use a for loop to help display tables of data. For example, suppose that we want to display how the landscape structure changes over time given the ecological succession model used in Example 8.1. We can use an `sprintf` statement inside of a loop to print out a table that displays this information. See Appendix A.5 for details on using the `sprintf` command.

EcoSuccessionTable.R

```
# Filename: EcoSuccessionTable.R
# R script to
# - Print out a table showing landscape structure over time

# A function to perform matrix exponentiation
"%^%"<-function(A,n){
  if(n==1) A else {B<-A; for(i in (2:n)){A<-A%*%B}}; A
  }

# Enter the transfer matrix
T = matrix(c( 0.94, 0.05, 0.01,
              0.02, 0.86, 0.12,
              0.01, 0.06, 0.93 ), ncol = 3)

# Enter the initial state
x0 = matrix(c( 1, 0, 0 ), ncol = 1)

# Print header for table
cat(" t      u      s      d\n")
cat("-------------------------\n")

# Fill in table using a for loop
for (t in c(1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 100, 200)){
   x = T%^%t %*% x0        # matrix multiplication
   u = x[1,1]              # get u value for this time step
   s = x[2,1]              # get s value for this time step
   d = x[3,1]              # get d value for this time step
   cat(sprintf("%3d  %5.3f  %5.3f  %5.3f",t,u,s,d),"\n")
   }
```

Observe that R does not have a built-in function to perform matrix exponentiation, hence the need for lines 6 – 8 above. When this R script is run in the Command Window, the output looks like

Command Window

```
> source("EcoSuccessionTable.R")
 t      u      s      d
-------------------------
  1  0.940  0.050  0.010
  2  0.885  0.091  0.025
  3  0.834  0.124  0.043
  4  0.787  0.151  0.063
  5  0.743  0.173  0.084
 10  0.569  0.236  0.194
 20  0.368  0.274  0.358
 30  0.273  0.284  0.444
 40  0.227  0.287  0.486
 50  0.205  0.289  0.506
100  0.185  0.291  0.524
200  0.184  0.291  0.525
```

Plotting Time Dynamics

Now that we can produce tables of times series data, it would be nice to transfer that information to a plot so that we can view the information graphically. We can do this by creating the time series data we want using a loop, and then using the `plot` command. The following R script plots times series data for the ecological succession model from Example 8.1 for time steps $t = 0$ to $t = 200$.

EcoSuccessionPlot.R

```
# Filename: EcoSuccessionPlot.R
# R script to
# - Print out a table showing landscape structure over time

# A function to perform matrix exponentiation
"%^%"<-function(A,n){
  if(n==1) A else {B<-A; for(i in (2:n)){A<-A%*%B}}; A
  }

# Enter the transfer matrix
T = matrix(c( 0.94, 0.05, 0.01,
              0.02, 0.86, 0.12,
              0.01, 0.06, 0.93 ), ncol = 3)

```

```
# Enter the initial state
x0 = matrix(c( 1, 0, 0 ), ncol = 1)

# We will create a matrix x that has three columns.
# Each column will contain time series data for one class.
# Each row will correspond to a time step.
x = matrix(rep(0, 201*3), ncol = 3)

x[1, ] = x0     # Data for time step t = 0

# Use for loop to generate times series data
for (t in 1:200){
   x[t+1, ] = T%^%t %*% x0 # Data for time step t
   }

# Time series information for proportion underwater is
# in the first column
u = x[ ,1]

# Time series information for proportion saturated but
# not underwater is in the second column
s = x[ ,2]

# Time series information for proportion dry is in the
# thid column
d = x[ , 3]

# Generate plot
time = seq(1, 200, by = 1)
plot(u,
   col  = "red",
   type = "l",
   xlab = "Time step t",
   ylab = "Proportion of Wetlands",
   xlim = c(1, 200),
   ylim = c(0, 1))
par(new = TRUE)
plot(s,
   col  = "green",
   type = "l",
   xlab = "",
   ylab = "",
   xlim = c(1, 200),
   ylim = c(0, 1),
   xaxt = "n",
   yaxt = "n")
```

```
par(new = TRUE)
plot(d,
   col  = "blue",
   type = "l",
   xlab = "",
   ylab = "",
   xlim = c(1, 200),
   ylim = c(0, 1),
   xaxt = "n",
   yaxt = "n")
legend("topright",
   c("u", "s", "d"),
   col = c("red", "green", "blue"),
   lty = c(1, 1, 1))
```

The plot resulting from this R script is shown in Figure 7.2.

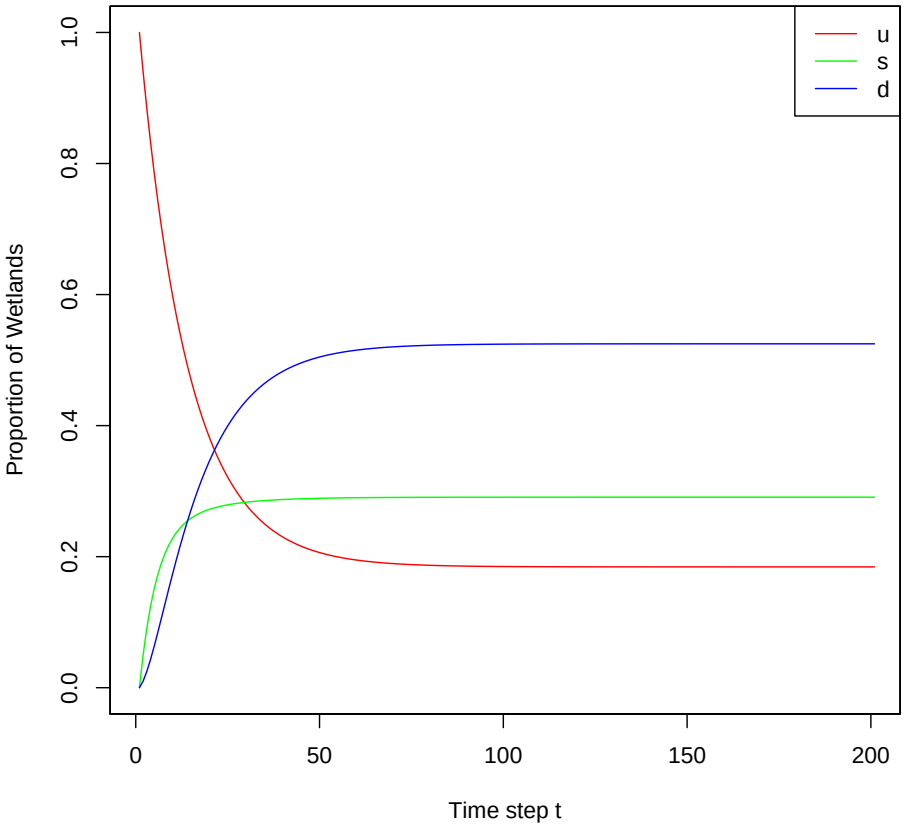


Figure 7.2: Plot generate by the R script EcoSuccessionPlot.R.

8.4 R Skills

In R we use the function `eigen` to find eigenvalues (which we will discuss in the following chapter) and eigenvectors. Since the output of this function will not be completely understood until the material of the next chapter is covered, we postpone demonstrating how to find eigenvectors in R until the end of the next chapter.

9.4 R Skills

In R, the `eigen` function calculates both the eigenvalues and their associated eigenvectors. If we have defined an $n \times n$ matrix `A`, then the command

```
eigen(A)
```

will produce a vector of length n containing eigenvalues, and an $n \times n$ matrix containing the associated eigenvectors. The commands

```
lambda = eigen(A)$values
v      = eigen(A)$vectors
```

assigns the eigenvalues and eigenvectors separately. The first column of `v` contains the eigenvector that corresponds with the eigenvalue shown in `lambda[1]`. The second column of `v` contains the eigenvector that corresponds with the eigenvalue shown in `lambda[2]`, and so on with the other columns of `v`.

Thus, if we wanted to find the eigenvalues and corresponding eigenvectors of

$$\mathbf{A} = \begin{bmatrix} 1 & 4 \\ 0.5 & 0 \end{bmatrix}$$

we could use the following commands in the Command Window

Command Window

```
> A = matrix(c(1, 0.5, 4, 0), ncol = 2)
> A
     [,1] [,2]
[1,]  1.0    4
[2,]  0.5    0
> eigen(A)
$values
[1]  2 -1

$vectors
          [,1]       [,2]
[1,] 0.9701425 -0.7844272
[2,] 0.2425356  0.4472136
> lambda = eigen(A)$values
> lambda
[1]  2 -1
> v = eigen(A)$vectors
> v
          [,1]       [,2]
```

```
[1,] 0.9701425 -0.8944272
[2,] 0.2425356  0.4472136
```

We see that the eigenvalues are $\lambda = \{2, -1\}$, which matches what we found in Example 9.3. The dominant eigenvalue is $\lambda = 2$ which is shown in `lambda[1]`. Note that R always displays the dominant eigenvalue in the first row, first column. We look to the first column of `v` to get the eigenvector corresponding to the dominant eigenvalue. R tells us this is

$$\begin{bmatrix} 0.9701 \\ 0.2425 \end{bmatrix}.$$

You might notice that this eigenvector does not match the one we found in Example 9.7. However, recall that eigenvectors are not unique, only the normalized eigenvector is unique. Let us normalize the eigenvector corresponding to the dominant eigenvalue. To do this in R, we divide the eigenvector by the sum of the entries in the eigenvector which we can obtain by using the function `sum`.

Command Window

```
> v[,1] / sum(v[,1])
[1] 0.8 0.2
```

Now, this normalized eigenvector matches the normalized eigenvector for the dominant eigenvalue we found in Example 9.7.

Now that we know how to find eigenvalues and normalized eigenvectors, we would like to see how eigenvectors change when we modify values in the transfer or Leslie matrices. For this, we will make use of loops.

Suppose we know that the Leslie matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 2 & 3 \\ 0.5 & 0 & 0 \\ 0 & 0.25 & 0 \end{bmatrix}$$

models a population that contains three age classes: hatchlings, juveniles, and adults. Suppose we wanted to explore how changing the juvenile fecundity changes the long term population structure (i.e. the eigenvector associated with the dominant eigenvalue). We can construct a for loop, to loop through the juvenile fecundity values we want to evaluate, and then display the results in a table or graph. Let us look at the following set of juvenile fecundity values,

$$a_{1,2} = \{0.1, 0.2, 0.5, 1.5, 2, 2.5, 3, 4\}.$$

Below is an R script that finds the dominant eigenvalue and corresponding normalized eigenvector for $\mathbf{A}$ given each juvenile fecundity value. The R script con-

structs a table of results, and plots the long term population structure against the juvenile fecundity.

LeslieFecundity.R

```
# Filename: LeslieFecundity.R
# R script to
#      - Find the dominant eigenvalue and corresponding
#        eigenvectors for an array of juvenile fecundities
#      - Print out a table of the results
#      - Print out results graphically

# Print out the header for the table
cat("Juvenile    Dominant        Proportion Structure    \n")
cat("Fecundity  Eigenvalue  Hatchlings  Juveniles  Adults\n")
cat("-------------------------------------------------------\n")

# Array of juvenile fecundity values
f = c(0.1, 0.2, 0.5, 1.5, 2, 2.5, 3, 4)

# Make a vector that we will use later
normv = t(t(numeric(3)))

# Start the loop
for (i in 1:length(f)){ # loop through length of f vector

   # Construct Leslie matrix with appropriate juvenile fecundity
   A = matrix( c( 0, 0.5, 0,
                  f[i], 0, 0.25,
                  3, 0, 0 ), ncol = 3)

   # Get eigenvalues and eigenvectors
   lambda = Re(eigen(A)$values)
   v = Re(eigen(A)$vectors)

   # Find dominant eigenvalue in lambda
   # and make note of position in lambda matrix
   lambdavector = abs(lambda) # array of eigenvalues
   for (j in 1:length(lambdavector)){
      if (max(abs(lambda)) == abs(lambdavector[j])){
         loc = j # position of dom eig in lambda vector
         deig = lambda[j] # domianant eigenvalue
         }
      }

   # Normalize eigenvector associated with dominant eigenvalue
   normv = cbind(normv, v[,loc]/sum(v[,loc]))
```

```
   # This creates a matrix called normv in which the
   # i^th column corresponds to the dominant eigenvector
   # associated with the i^th juvenile fecundity value

   # Print these values
   cat(sprintf("   %3.1f   ",f[i])) # juvenile fecundity value
   cat(sprintf("     %5.3f   ",deig)) # dominant eigenvalue
   cat(sprintf("   %5.3f  ",normv[1,i+1])) # hatchling prop. at eq
   cat(sprintf("     %5.3f  ",normv[2,i+1])) # juv. prop. at eq
   cat(sprintf("   %5.3f  ",normv[3,i+1])) # adult prop. at eq
   cat("\n") # new line
   }

# Remove the first column from normv
normv = normv[,-1]

# Generate graph
h = normv[1,] # vector of hatchling proportions at eq
j = normv[2,] # vector of juvenile proportions at eq
a = normv[3,] # vector of adult proportions at eq

plot(f, h,
   col = "red",
   type = "b",
   pch = 20,
   xlab = "Juvenile Fecundity",
   ylab = "Equilibrium Structure",
   xlim = c(0, 4),
   ylim = c(0, 1))
par(new = TRUE)
plot(f, j,
   col = "green",
   type = "b",
   pch = 20,
   xlab = "",
   ylab = "",
   xaxt = "n",
   yaxt = "n",
   xlim = c(0, 4),
   ylim = c(0, 1))
par(new = TRUE)
plot(f, a,
   col = "blue",
   type = "b",
   pch = 20,
   xlab = "",
```

```
   ylab = "",
   xaxt = "n",
   yaxt = "n",
   xlim = c(0, 4),
   ylim = c(0, 1))
legend("topright",
   c("Hatchlings", "Juveniles", "Adults"),
   col = c("red", "green", "blue"),
   lty = c(1,1,1),
   pch = c(20, 20, 20))
```

When this R script is run in the Command Window, the output looks like

Command Window

```
> source("LeslieFecundity.R")
Juvenile    Dominant        Proportion Structure
Fecundity  Eigenvalue  Hatchlings  Juvenile  Adults
-----------------------------------------------------
  0.1        0.744       0.527       0.354    0.119
  0.2        0.767       0.536       0.350    0.114
  0.5        0.836       0.563       0.337    0.101
  1.5        1.052       0.630       0.299    0.071
  2.0        1.151       0.654       0.284    0.062
  2.5        1.245       0.675       0.271    0.054
  3.0        1.335       0.692       0.259    0.049
  4.0        1.500       0.720       0.240    0.040
```

The graphical output of this R script is shown in Figure 9.1.

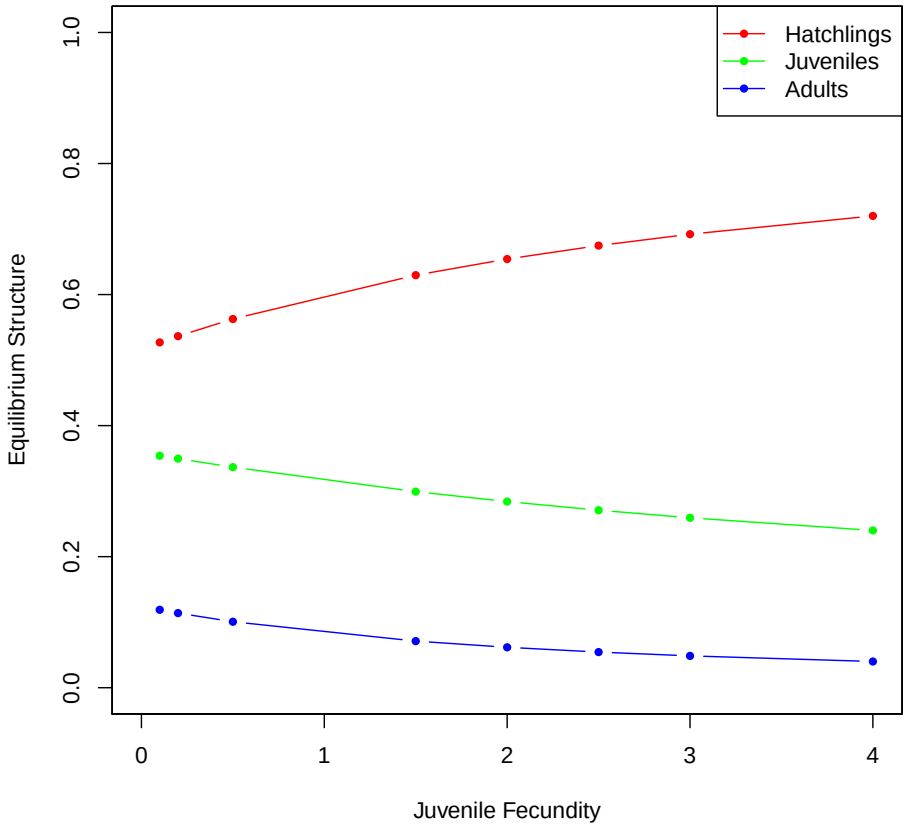


Figure 9.1: Plot generated from the R script LeslieFecundity.R.

10.5 R Skills

Given a certain experiment $\mathcal{E}$, we would like to utilize R to find the probability of an event $E \subset S$, where S is the sample space for the experiment $\mathcal{E}$. To do this, we will write code in R that simulates the experiment. Then we will repeat the experiment many times, noting the outcome of the experiment each time. If we run the experiment 1000 times and the outcomes corresponding to event E occur 126 times, then we will say $P(E) = 126/1000 = 0.126 = 12.6\%$.

Generating Random Numbers

Before we can successfully simulate an experiment, we need a way to introduce randomness or unpredictability into our simulations. R can produce "random" numbers that you can think of as arising by tossing a dart at the line segment between 0 and 1, with the dart having equal chance of landing at any point. That is, you don't have good aim, so the dart can hit near the ends near 0 just as likely as it hits in the middle near 0.5 or near the end at 1. R uses a numerical method to do this that is good, but not perfect, and so officially these are called "pseudorandom numbers." For our purposes, we'll just abbreviate this and call them random numbers. The function `runif(x)`, where x is a positive integer, will return a vector of length x containing random numbers between 0 and 1, that is in the open interval (0, 1). These numbers are said to be chosen according to a "uniform random distribution" on the interval (0, 1). Thus, `rand(1)` will return a single random number between 0 and 1, and `rand(4)` will return a vector with 4 entries, each being a random number between 0 and 1. If you want to construct an $n \times m$ matrix containing random numbers, use

```
matrix(runif(n*m), ncol = m)
```

Suppose we do not want random numbers between 0 and 1. Table 10.2 shows some examples of how to modify the `runif` function in order to obtain random numbers in various intervals.

Often when we are conducting an experiment we want to choose between one of several possible outcomes. For example, if we are flipping a coin there are two possible outcomes. If we are rolling a standard die there are six possible outcomes. To pick among the various choices we will use a combination of the `runif` function, the `ceiling` and `floor` functions, and a code structure known as an *if statement*. An *if statement* has the following format

```
if (condition){
   # what to do if condition is true
   }
else
```

Interval	Code
(0,1)	`runif(1)`
(1,2)	`runif(1) + 1`
(-1, 0)	`runif(1) - 1`
(0, 10)	`10*runif(1)`
(0, 0.5)	`(0.5)*runif(1)`
(-1,1)	`2*runif(1) - 1`
(-10, 10)	`2*10*runif(1) - 10`

Table 10.2: The R code in the right hand column will produce a random number in the corresponding interval listed in the left hand column.

```
# what to do if condition is false
}
```

Suppose you wanted to simulate the flipping of a coin. You want to use the function `runif` to generate two possibilities. First, we generate a random number, `x`, in the interval (0, 2). Next, take `floor(x)` to round the value down. If the `x` is in the interval (0,1), `floor` will round it down to 0. If the `x` is in the interval [1,2), `floor` will round it down to 1. Next, we can use the *if statement* to say that if `floor(x)` is equal to 0, then it is a heads. Otherwise, if `floor(x)` is equal to 1, then it is a tails. Here is what that code would look like.

coinflip.R

```
# R script for flipping a coin

# Generate a random number in (0,2)
x = 2*runif(1)

# Determine if heads or tails
if (floor(x) == 0) {
   cat("Your coin landed head side up!\n")
   } else {
   cat("Your coin landed tail side up!\n")
   }
```

Notice that to test the equality of the condition that `floor(x)` is equal to zero we use a double equals sign, ==. This notation is used to differentiate between a test for equality (as we did above in the *if statement*) and setting a variable equal to a value (as we did when we set `x` equal to a random number between 0 and 2). Try copying this R script and then running it several times to see that sometimes you get heads and other times you get tails.

Suppose you wanted to simulate rolling a die. Here we will not need to use the *if statement* since all we need is the generated number.

dieroll.R

```
# R script for rolling a die

# Generate a random number in (0,6)
x = 6*runif(1)

# Print out result
cat(sprintf("You rolled a %d.", ceiling(x)), "\n")
```

Again, try copying this R script and then running it several times to see that you roll different values. Notice in the R script for rolling the die, we used the function `ceiling` instead of `floor`. This is because we wanted to round up to the values 1, 2, 3, 4, 5, and 6. If we had used `floor` the random number would have been rounded down to one of the values 0, 1, 2, 3, 4, or 5.

Writing an R script as a Function

As we will see shortly, it is often valuable to write our own functions in R. These functions will work the same way as the built-in functions in R, like `predict`, `floor`, `mean`, etc. However, when we write our own functions we can have as many inputs and outputs as we want, and we can define the function to perform any number of operations desired. Functions in R are R scripts where the first line of executable code starts with the word `function`.

Suppose you wanted to flip a coin n times, where n is some integer value. We could write this as a function where the input is the number of times you wish to flip the coin, and the output is a 1×2 vector that contains the number of times heads was flipped and the number of times tails was flipped. The R script for this function is shown below and is called coinflips.R.

coinflips.R

```
# function outcome = coinflips(N)
#
# Input:
#  N = number of times to flip the coin
#
# Output:
#  outcome = array with structure
#    [# heads flipped, # tails flipped]
#
coinflips = function(N){

```

```
12    # Get N random numbers in (0, 2)
13    x = 2*runif(N)
14
15    # Initialize heads and tails counters to zero
16    Hcount = 0
17    Tcount = 0
18
19    # For each random number, decide if it is a head or tail
20    for (i in 1:N){
21       if (floor(x[i]) == 0) {
22          Hcount = Hcount + 1
23          } else {
24          Tcount = Tcount + 1
25          }
26       }
27
28    # Record outcome
29    outcome = c(Hcount, Tcount)
30    return(outcome)
31    }
```

Notice that we create a vector of `N` random numbers (`N` being the input of the function), one for each coin flip. Since we want to keep track of how many times a head is flipped and how many times a tail is flipped, we create counters called `Hcount` and `Tcount` and set them initially to zero. Next, we use a loop to go through each of the coin flips. For each coin flip, we use an *if statement* to decide whether it was a head or tail that was flipped. After we have considered each coin flip (i.e. after we are done with the loop) we construct our output vector, `outcome`.

Suppose we wanted to simulate 1000 coinflips. Now that we have our `coinflips(N)` function, we can simply type into the command window

Command Window

```
1 > source("coinflips.R")
2 > coinflips(1000)
```

If we run this same command multiple times, the output of the function will vary each time. Below shows the output when one of the authors ran this command 10 times.

Command Window

```
1 > coinflips(1000)
2 [1] 499 501
3 > coinflips(1000)
```

```
[1] 503 497
> coinflips(1000)
[1] 497 503
> coinflips(1000)
[1] 497 503
> coinflips(1000)
[1] 491 509
> coinflips(1000)
[1] 512 488
> coinflips(1000)
[1] 498 502
> coinflips(1000)
[1] 503 497
> coinflips(1000)
[1] 492 508
> coinflips(1000)
[1] 507 493
```

Estimating the Probablity of an Event

Once we have a way of simulating a certain experiment multiple times, we would like to estimate the probability of a certain event occurring.

Returning to the coin flipping example, suppose we wanted to know the probability of flipping a head. We could run our `coinflips` function with $N = 1000$, see how many times heads occurred and then divide by 1000. However, if we did this, we would get a slightly different probability each time. For example, the 10 outputs shown above would correspond to the probabilities 0.499, 0.503, 0.497, 0.497, 0.491, 0.512, 0.498, 0.503, 0.492, and 0.507 (for flipping a heads).

Another approach to estimating the probability of flipping a heads would be to take the average of the 10 different outputs and divide that number by 1000. If we did this we would get a probability of 0.4999. We could extend this and take the average of 500 different outputs and divide that number by 1000. However, we would not want to enter `coinflips(1000)` into the Command Window 500 times. Thus, let us write an R script that does this for us.

ProbHeadFlip.R

```
# Filename: ProbHeadFlip.R
# R script to compute probability of flipping a heads

# Number of times to collect output
n = 500

# Initialize heads count
```

```
Hcount = 0

# Run coinflips n times
for (i in 1:n){
   # Coinflips will run 1000 experiments each time
   output = coinflips(1000)

   # sum up number of heads outputs
   Hcount = Hcount + output[1]
   }

# Compute average number of heads
Haverage = Hcount / 1000

# Estimate probability of flipping heads
ProbH = Haverage / n

# Print out answer
cat(sprintf("P(Heads) = %6.4f", ProbH), "\n")
```

When one of the authors ran this file five times in the Command Window, the outputs were as follows.

Command Window

```
> source("ProbHeadFlip.R")
P(Heads) = 0.4986
> source("ProbHeadFlip.R")
P(Heads) = 0.4990
> source("ProbHeadFlip.R")
P(Heads) = 0.4990
> source("ProbHeadFlip.R")
P(Heads) = 0.5000
> source("ProbHeadFlip.R")
P(Heads) = 0.5011
```

We can see that these values are a better estimate of the true probability of flipping heads, $P(H) = 0.5$, than if we just used one output from the `coinflips` function with $N = 1000$.

If we change the value of n in the `ProbHeadFlip.R` file to $n = 10000$, the answers will be even closer to the true probability. However, the larger we make n, the longer the file will take to run and at some point we must decide that our estimate is good enough.

Biological Example of a Probability Estimation:

Albinism

Here we will consider an example similar to Example 10.7.

Suppose we know that John and Jane are both carriers for albinism. What is the probability that their child will be a carrier for albinism? What is the probability that a child of theirs will have albinism?

Simulate Generation of an Offspring Genotype

We can use R to simulate the "experiment" of John and Jane having a child. First, we construct a function in R where the input is the genotype of each parent (with respect to albinism), and the output is the genotype of the offspring. For the genotype with respect to albinism, there are three possibilities: (1) heterozygous (Aa or aA), (2) homozygous dominant (AA), or (3) homozygous recessive (aa). In our R function, we will use numbers to represent each of these cases.

OneChild.R

```
# function child = OneChild(mom,dad)
#
# Inputs:
#   mom = genotype of mother**
#   dad = genotype of father**
# ** For genotypes use
#     1 for heterozygous, Aa or aA
#     2 for homozygous dominant, AA
#     3 for homozygous recessive, aa
#
# Output:
#   child = genotype of offspring
OneChild = function(mom,dad){

# Within this function we will use
#   0 to represent a recessive allele
#   1 to represent a dominant allele

# Determine allele inherited from mother
if (mom == 1) {
   # if mom heterozygous, randomly choose between two alleles
   childallele1 = floor(2*runif(1))
   } else if (mom == 2) {
   # if mom homozygous dominant, then child inherits A
   childallele1 = 1
   } else {
```

```
   # if mom homozygous recessive, then child inherits a
   childallele1 = 0
   }

# Determine allele inherited from father
if (dad == 1) {
   # if dad heterozygous, randomly choose between two alleles
   childallele2 = floor(2*runif(1))
   } else if (dad == 2) {
   # if dadm homozygous dominant, then child inherits A
   childallele2 = 1
   } else {
   # if dad homozygous recessive, then child inherits a
   childallele2 = 0
   }

# Determine the genotype of the child
if ((childallele1 == 1) && (childallele2 == 1)) {
   child = 2
   } else if ((childallele1 == 0) && (childallele2 == 0)) {
   child = 3
   } else {
   child = 1
   }

# Return the genotype
return(child)
}
```

Notice in the `OneChild` function the use of `else if` in the *if statements* to test a second condition. In this case, when determining the allele inherited from the mother, first the code checks if the mother is heterozygous. If she is then we use the `runif` function to determine the allele that is inherited from the mother. If she is not heterozygous, then the code goes to the next condition and checks if the mother is homozygous dominant. If she is then the child will inherit a dominant allele. Lastly, if the mother is not heterozygous and not homozygous dominant then the code goes to the else condition. This assumes that if the mother is not heterozygous and not homozygous dominant then the only option left is that she is homozygous recessive, in which case the child inherits a recessive allele. This process is repreated from choosing an allele from the father.

Next, notice when the genotype of the child is determined (in the third and last *if statement* structure), we use the `&&` notation to indicate that the condition in the *if statement* that must be satisfied is actually two conditions that must both be satisfied. We use the `||` notation to indicate that the condition in the

if statement is true if either of the conditions are satisfied. So, if our code reads something like

```
if (condition A && condition B) {
   # something here
   } else {
   # something here
   }
```

then the *if statement* is true if condition A AND condition B hold. Likewise, if our code reads something like

```
if (condition A || condition B) {
   # something here
   } else {
   # something here
   }
```

then the *if statement* is true if condition A OR condition B hold.

Estimating Probabilities

Now that we have a function for the "experiment" of generating the genotype of an offspring, we would like to calculate the probability of different events. We will find these probabilities by running the experiment 1000 times, tabulating the results, and then taking the average of 1000 of the 1000 experiment runs. This is similar to how we estimated the probability of flipping a heads in the coin flipping experiment. Below is the code to complete these estimations, named `Albinism.R`.

`Albinism.R`

```
1  # Filename: Albinism.R
2  # R script for determining the probability of child with two
3  #   albinism carrying parents
4  # The possible outcomes for the child are
5  #   (a) an albinism carrier (Aa, aA)
6  #   (b) have no albinism alleles (AA)
7  #   (c) albino (aa)
8  # This file finds the probability of each outcome.
9
10 # Set how many times to run the experiment
11 N = 1000
12
13 # Set variables to sum up probabilities
```

```
A = 0      # sum of heterozygous probabilities
B = 0      # sum of homozygous dominant probabilities
C = 0      # sum of homozygous recessive probabilities

for (i in 1:N) {
   # Set counters
   a = 0   # heterozygous counter
   b = 0   # homozygous dominant counter
   c = 0   # homozygous recessive probabilities

   for (j in 1:N) {
      # Get child's genotype if both parents are carriers
      child = OneChild(1,1)

      # Increase appropriate counter
      if (child == 1) {
         a = a + 1
         } else if (child == 2) {
         b = b + 1
         } else {
         c = c + 1
         }
      }

   # Add the probabilities for this set of N experiments
   # to the running sum
   A = A + a/N
   B = B + b/N
   C = C + c/N
   }

# Print out results
cat(sprintf("P(carrier of albinism) = %6.4f", A/N), "\n")
cat(sprintf("P(has albinism) = %6.4f", C/N), "\n")
cat(sprintf("P(no recessive alleles) = %6.4f", B/N), "\n")
```

When one of the authors ran this file five times in the Command Window, the outputs were as follows.

Command Window

```
> source("Albinism.R")
P(carrier of albinism) = 0.5003
P(has albinism) = 0.2496
P(no recessive alleles) = 0.2501
> source("Albinism.R")
```

```
P(carrier of albinism) = 0.4994
P(has albinism) = 0.2497
P(no recessive alleles) = 0.2509
> source("Albinism.R")
P(carrier of albinism) = 0.5002
P(has albinism) = 0.2502
P(no recessive alleles) = 0.2496
> source("Albinism.R")
P(carrier of albinism) = 0.5002
P(has albinism) = 0.2502
P(no recessive alleles) = 0.2496
> source("Albinism.R")
P(carrier of albinism) = 0.5003
P(has albinism) = 0.2498
P(no recessive alleles) = 0.2499
```

11.4 R Skills

Estimating the Probablity of Compound Events

Estimating the probability of compound events is similar to estimating the probability of a single event, only now we must keep track of multiple events.

Consider Example 11.9 where two dice were rolled and we want to know the probability of rolling doubles of a sum of 6. We can easily simulate the rolling of two dice, however, now we must keep track of every time we roll doubles (i.e. the two numbers rolled are equal) and every time we roll a sum of 6. Otherwise, the probability of the event "doubles or sum of 6" is computed in the same fashion as in Section 10.5. Below is an example of an R script that computes the probability of rolling doubles or a sum of 6.

DiceRolling.R

```
# Filename: DiceRolling
# R script to compute probability of rolling doubles or
#    a sum of 6.

# Number of times to run experiment
N = 1000

# Initialize sum of probabilities
C = 0

for (i in 1:N) {
   # Set event counter to zero
   c = 0

   for (j in 1:N) {
      # Roll two dice x and y
      x = ceiling(6*runif(1))
      y = ceiling(6*runif(1))

      # If roll doubles or sum 6, increase counter.
      # Otherwise do nothing
      if ((x == y) || (x + y == 6)) {
         c = c + 1
         }
      }

   # Add on prob of event after rolling N times
   C = C + c/N
   }

```

```
# Print out result
cat(sprintf("P(doubles OR sum of 6) = %6.4f", C/N), "\n")
```

Generating Genotypes with Multiple Allele Pairs

In this chapter we looked at genetic traits that are determined by more than one gene. For example, blood type with Rh-factor (one gene for blood type, another gene for Rh-factor), and eye color (a brown/blue gene and a blue/green gene, for more on eye color genese see Section 14.3 for the Eye Color Project).

Another example of this comes from the classical pea plant experiments by Gregor Mendel. Mendel did many of his experiments on the pea plant *Pisum sativum*. Within these experiments he traced the traits of many traits through several generations of pea plants. Two of those traits were whether the seeds produced were spherical or wrinkled and whether the seeds produced were green or yellow. As it turns out, these two traits for *Pisum sativum* seeds are determined by two separate genes. The genese for spherical/wrinkled seeds can have S alleles (dominant) for spherical seeds and s alleles (recessive) for wrinkled seeds. The genes for yellow/green seeds can have Y alleles (dominant) for yellow seeds and y alleles (recessive) for green seeds.

If we want to determine the probability of an event such as $E =$ "offspring has yellow wrinkled seeds" we need to be able to simulate the experiment of two parent plants (each with two seed genese) producing an offspring with two seed genes, assuming that the two genes are independent. Below is the R function `OneChild2Genes` that completes this task. This function is similar to the `OneChild` function in Section 10.5. The inputs are the genotypes of each parent (for both genes), and the output is the genotype of the offspring.

`OneChild2Genes.R`

```
# function child = OneChild2Genes(mom,dad)
#
# Inputs:
#   mom = genotype of mother**
#     array containing genotype of first & second genes
#   dad = genotype of father**
#     array containing genotype of first & second genes
# ** For genotypes use
#     1 for heterozygous, Aa or aA
#     2 for homozygous dominant, AA
#     3 for homozygous recessive, aa
#
# Output:
#   child = genotype of offspring array containing
#     genotype for first gene and second gene
```

```
16
17 OneChild2Genes = function(mom, dad) {
18
19 # Within this function we will use
20 #   0 to represent a recessive allele
21 #   1 to represent a dominant allele
22
23 # Declare an array to store the genotypes
24 child = numeric(2)
25
26 # Loop through both genes
27 for (i in 1:2) {
28    # determine allele for gene i from mother
29    if (mom[i] == 1) {
30       # if mom heterozygous, randomly choose between two alleles
31       childallele1 = floor(2*runif(1))
32       } else if (mom[i] == 2) {
33       # if mom homozygous dominant, then child inherits A
34       childallele1 = 1
35       } else {
36       # if mom homozygous dominant, then child inherits a
37       childallele1 = 0
38       }
39
40    # determine allele for gene i from father
41    if (dad[i] == 1) {
42       # if dad heterozygous, randomly choose between two alleles
43       childallele2 = floor(2*runif(1))
44       } else if (dad[i] == 2) {
45       # if dad homozygous dominant, then child inherits A
46       childallele2 = 1
47       } else {
48       # if dad homozygous dominant, then child inherits a
49       childallele2 = 0
50       }
51
52    # Determine the genotype of the child for gene i
53    if ((childallele1 == 1) && (childallele2 == 1)) {
54       child[i] = 2
55       } else if ((childallele1 == 0) && (childallele2 == 0)) {
56       child[i] = 3
57       } else {
58       child[i] = 1
59       }
60    }
61
```

```
62 # Return statement
63 return(child)
64 }
```

Notice the differences in this function from the `OneChild` function. The function `OneChild2Genes` now takes in 1×2 vectors of genotype information for each parent and then loops through each gene. The code contained inside the loop is the same as the code contained in `OneChild` except in `OneChild2Genes` the i^{th} gene is dealt with through each pass of the loop.

12.3 R Skills

To estimate the conditional probability $P(A|B)$ of an experiment $\mathcal{E}$, we simulate the experiments (as in Chapters 10 and 11), but now we must keep track of two events, $A \cap B$ and B, so that we can compute

$$P(A|B) = \frac{P(A \cap B)}{P(B)}.$$

Biological Example of Conditional Probability: Tay-Sachs Disease

Consider Example 12.1 where Judy has a little brother who has Tay-Sachs disease and is worried that she might be a carrier of the disease. From the example, we know that each of Judy's parents is a carrier for Tay-Sachs disease.

Generating the Offspring's Genotype

We can use the `OneChild` function developed in Section 10.5 to simulate the "experiment" of the generation of Judy's genotype. Recall the function `OneChild` takes in two integer values from the set $\{1, 2, 3\}$ representing the genotype of the mother and the father where

- 1 = heterozygous genotype, Aa or aA
- 2 = homozygous dominant, AA, and
- 3 = homozygous recessive, aa.

The output of the file is an integer value from the set $\{1, 2, 3\}$ which represents the output of the offspring.

Computing the Conditional Probability

Using the `OneChild` function we can repeat the experiment many times and compute the probability. However, now we must keep track of how many times two different events occur. Let A = "Judy is heterozygous" and $\overline{B}$ = "Judy is not homozygous recessive." Notice $A \cap B = A$. Thus,

$$P(A|\overline{B}) = \frac{P(A \cap \overline{B})}{P(\overline{B})} = \frac{P(A)}{P(\overline{B})}.$$

Therefore, in order to compute the probability that Judy is a carrier given that she does not have the disease, we must keep track of the number of experiments

that produce a heterozygous genotype and the number of experiments that produce a homozygous recessive genotype. The code to do this is shown below and is called `TaySachs.R`.

TaySachs.R

```
# Filename: TaySachs.R
# R script for determining P(A|not B) where
#   A = offspring is a carrier
#   B = offspring does have TaySachs
# given that both parents are carriers

# We'll need the function from OneChild.R
source("OneChild.R")

# Set how many times to run the experiment
N = 1000

# Set variables to sum up the probabilities of events
A = 0       # probability of Aa or aA
B = 0       # probability of aa

# Run set of N experiments N times
for (i in 1:N) {
   # Set counters
   a = 0    # heterozygous counter
   b = 0    # homozygous recessive counter

   # Run experiment N times
   for (j in 1:N) {
      # Get child's genotype if both parents are carriers
      child = OneChild(1,1)

      # Increase appropriate counter
      if (child == 1) {
         a = a + 1
         } else if (child == 3) {
         b = b + 1
         } else {
         # do nothing
         }
      }

   # Add the probabilities for this set of N experiments
   # to the running sum
   A = A + a/N
   B = B + b/N
```

```
   }

# Estimated Probability of Event A
PA = A/N

# Estimated Probability of Event NOT B
PB = 1 - B/N

# Print out results
cat(sprintf("P(carrier|no disease)=%6.4f", PA/PB), "\n")
```

Notice that since the code inside of the loops kept track of when a homozygous recessive genotype was generated and we wanted the probability of when this did not happen, at the end we calculated

$$P(\overline{B}) = 1 - P(B)$$

where $B =$ "Judy is homozygous recessive."

When one of the authors ran `TaySachs.m` five times in the Command Window, the outputs were as follows.

Command Window

```
> source("TaySachs.R")
P(carrier|no disease)=0.6661
> source("TaySachs.R")
P(carrier|no disease)=0.6668
> source("TaySachs.R")
P(carrier|no disease)=0.6669
> source("TaySachs.R")
P(carrier|no disease)=0.6670
> source("TaySachs.R")
P(carrier|no disease)=0.6665
```

Biological Example of Conditional Probability: Drug Testing

Recall Example 12.2 where a new sleeping pill is being tested. Of the 200 individuals participating in the study, 100 are given that new sleeping pill and the reamining 100 are given a sugar pill (a placebo). The results of the study are given in the following table.

In Example 12.2 we ask what is the probabiity that if you take the sleeping pill you will sleep better? Let $A =$ "took sleeping pill" and $B =$ "slept better." We

	Improved sleep	Did not sleep better
Sleeping pill	71	29
Sugar Pill	58	42

want to estimate

$$P(B|A) = \frac{P(A \cap B)}{P(A)}.$$

To do this, first we need to simulate an experiment where we randomly choose one of the 200 study participants and then check whether they took the sleeping pill or sugar pill, and whether they had improved sleep or not. Once we have an R function to simulate this experiment, we can repeat this experiment several times, keeping track of the events $A \cap B$, "took sleeping pill AND had improved sleep" and A, "took sleeping pill." By keeping track of these events we can estimate the probability of sleeping better given that you took the sleeping pill.

Simulating the Experiment

To simulate the experiment we will construct a 200×2 matrix where each row represents an individual who participated in the study, the first column represents whether they took a sugar pill or sleeping pill, and the second column represents whether they had improved sleep or not. In the first column, we will use 0 to represent took sugar pill, and 1 to represent took sleeping pill. In the second column, we will use 0 to represent did not have improved sleep, and 1 to represent did have improved sleep. Note that this R function will have no input. However, the output will be a 1×2 matrix, specifically the row of the 200×2 matrix representing the individual who was selected.

DrugTesting.R

```
# function out = DrugTesting
# Inputs: none
#
# Output:
#   out = 1x2 matrix representing results for 1 individual
#     column 1: 0 = sugar pill, 1 = sleeping pill
#     column 2: 0 = no improved sleep, 1 = improved sleep

DrugTesting = function() {

# Create matrix of drug testing results
Results = matrix(rep(0, 200*2), ncol = 2)

# Set first 1/2 of column 1 to "took sleeping pill"
Results[1:100,1] = 1
```

```
# Set first 71 of column 2 to "improved sleep"
Results[1:71,2] = 1
# Now rows 1-71 = "took sleeping pill" & "had improved sleep"
# and rows 72-100 = "took sleeping pill" & "no improved sleep"

# Set rows 101-158 of column to "improved sleep"
Results[101:158,2] = 1
# Now rows 101-158 = "took sugar pill" & "had improved sleep"
# and rows 159-200 = "took sugar pill" & "no improved sleep"

# Randomly pick one of the 200 individuals
x = ceiling(200*runif(1))  # generates intengers from 1 to 200

# Create function output
return(Results[x,])
}
```

Notice, the `DrugTesting` function is equally likely to choose any of the 200 participants.

Computing the Conditional Probabilities

Now that we have a function to conduct the experiment of choosing the drug study participants and displaying their results, we can replicate this experiment many times and keep track of the results to estimate various probabilities. We will write an R script to estimate the probability that a participant had improved sleep given that they took the sleeping pill.

`SleepingPillEfficacy.R`

```
# Filename: SleepingPillEfficacy.R
# R script to estimate probability of sleeping better given
#   that you take a sleeping pill

# Set how many times to run the experiment
N = 1000

# Set variables to sum up the probabilities of events
A = 0    # probability took sleeping pill
B = 0    # probability took sleeping pill + had improved sleep

# Run set of N experiments N times
for (i in 1:N) {
   # Set counters
   a = 0   # took sleeping pill counter
   b = 0   # took sleeping pill + had improved sleep counter
```

```
17
18    # Run experiment N times
19    for (j in 1:N) {
20       # Select one participant
21       x = DrugTesting()
22
23       # If participant took sleeping pill
24       if (x[1] == 1) {
25          a = a + 1
26
27          # If participant also had improved sleep
28          if (x[2] == 1) {
29             b = b + 1
30             }
31          }
32       }
33
34    # Add the probabilities for this set of N experiments
35    # to the running sum
36    A = A + a/N
37    B = B + b/N
38    }
39
40 # Print out result
41 cat(sprintf("P(improved sleep|took sleeping pill)
42    = %6.4f", (B/N)/(A/N)), "\n")
```

Notice the nested if statements. First, we check to see if the participant selected took the sleeping pill. If they did, then we increase the "took sleeping pill" counter and check if they also had improved sleep. If they also had improved sleep, then we additionally increase the "took sleeping pill + had improved sleep" counter.

When one of the authors ran `SleepingPillEfficacy.R` five times in the Command Window, the outputs were as follows.

Command Window

```
1 > source("SleepingPillEfficacy.R")
2 P(improved sleep|took sleeping pill)
3    = 0.7098
4 > source("SleepingPillEfficacy.R")
5 P(improved sleep|took sleeping pill)
6    = 0.7106
7 > source("SleepingPillEfficacy.R")
8 P(improved sleep|took sleeping pill)
9    = 0.7101
```

```
> source("SleepingPillEfficacy.R")
P(improved sleep|took sleeping pill)
   = 0.7106
> source("SleepingPillEfficacy.R")
P(improved sleep|took sleeping pill)
   = 0.7094
```

15.3 R Skills

If we have a definition of a function f, R can calculate the value of f at a specific value x. Recall, to define a "function" in R we must create an R script that defines that function. For example, if we needed to work with the function

$$f(x) = \frac{\sqrt{x^2+9}-3}{x^2},$$

we could create the following R script to define that function.

f.R

```
# Creates the function f(x)
# Input: x (a real number)
# Output: y

f = function(x) {
return((sqrt(x^2+9)-3)/(x^2))
}
```

Now, suppose we want to find the limit of $f(x)$ as $x \to 2$ or as $x \to 0$. With a program like R, we cannot find the exact limit, however we can estimate that limit numerically. Notice for

$$\lim_{x\to 0} \frac{\sqrt{x^2+9}-3}{x^2},$$

though a limit exists (which we showed in Example 15.3), the value $f(0)$ does not exist. Thus, to find the limit, we will look at the values of $f(x)$ for x close to 0 but $x \neq 0$. We can approximate the limit by looking at values $f(x+h)$ and $f(x-h)$ where h becomes increasingly small. Additionally, we will consider values close to $x = 0$ on both sides of $x = 0$ (see right and left limits in Chapter 16).

limit.R

```
# Find the limit of a function f
# Make sure f.m is contained in the same folder as this file

# Input: x = value at which you want to find the limit
#        delta = how close do you want to get to the value x

# Output: LL = limit from the left side
#         RL = limit from the right side

# Make sure we have f.R loaded
source("f1.R")
```

```
12
13 limit = function(h,x) {
14
15 # Left limit
16 LL = f(x - h)
17
18 # Right limit
19 RL = f(x + h)
20
21 return(c(LL, RL))
22 }
```

Now, if we want to estimate the limits as $x \to 0$, we would use the following commands in the Command Window,

Command Window

```
1 > source("limit.R")
2 > limit(1,0)
3 [1] 0.1622777 0.1622777
4 > limit(0.1,0)
5 [1] 0.1666204 0.1666204
6 > limit(0.01,0)
7 [1] 0.1666662 0.1666662
8 > limit(0.001,0)
9 [1] 0.1666667 0.1666667
```

Notice as we let the input for `h` get smaller and smaller the left and right limits seem to approach 0.16666... or $\frac{1}{6}$. Thus, given the numerical approximations, we estimate the limit as $x \to 0$ to be $\frac{1}{6}$.

Now, if we want to find the limits as $x \to 2$, we would use the following commands in the Command Window,

Command Window

```
1 > source("limit.R")
2 > limit(1,2)
3 [1] 0.1622777 0.1380712
4 > limit(0.1,2)
5 [1] 0.1526471 0.1501058
6 > limit(0.01,2)
7 [1] 0.1515148 0.1512606
8 > limit(0.001,2)
9 [1] 0.1514005 0.1513751
10 > limit(0.0001,2)
11 [1] 0.1513891 0.1513865
```

Notice as we let the input for `h` get smaller and smaller the left and right limits approach 0.1514. Thus, given these numerical approximations, we estimate the limit as $x \to 2$ to be 0.1514.

If we now want to consider a different function, like

$$f(x) = \frac{1}{(x-2)^2}$$

all we need to do is change the file `f.R` appropriately.

`f.R`

```
# Creates the function f(x)
# Input: x (a real number)
# Output: y

f = function(x) {
return(1/(x-2)^2)
}
```

Now, what do we find as $x \to 2$?

Command Window

```
> limit(1,2)
[1] 1 1
> limit(0.1,2)
[1] 100 100
> limit(0.01,2)
[1] 10000 10000
> limit(0.001,2)
[1] 1e+06 1e+06
> limit(0.0001,2)
[1] 1e+08 1e+08
> limit(0.00001,2)
[1] 1e+10 1e+10
```

Here it appears that the smaller we make the value of `h` the larger the value of $f(x)$ becomes. Thus, as $x \to 2$ we see that

$$\frac{1}{(x-2)^2}$$

is growing without bound. For this function, it would be a fair estimation to say the limit does not exist as $x \to 2$.

16.4 R Skills

Recall that if we want to work with the function $f(x) = x^2$ in R, we can write an R script like `f.R` below.

`f.R`

```
# Creates the function f(x)
# Input: x (a real number)
# Output: y

f = function(x) {
return(x^2)
}
```

Now, if we wanted to graph that function we could write another R script like `graphf.R` below.

`graphf.R`

```
# Create a graph of f
# Make sure f.R is contained in the same folder as this file
# Inputs: xmin = minimum x value to graph
#         xmax = maximum x value to graph
# Output: a graph

# Load f.R
source("f.R")

graphf = function(xmin,xmax) {

# Create 1000 x values to graph
n = (xmax - xmin) / 1000
x = seq(xmin, xmax, by = n)

# Construct an array to hold f(x) values
F = numeric(length(x))

# Create 1000 f(x) values to graph
for (i in 1:length(x)) {
   F[i] = f(x[i])
   }

# Create plot
plot(x, F,
   col = "red",
   type = "l",
```

```
28     xlab = "x",
29     ylab = "f(x)")
30 }
```

When run in the Command Window using

Command Window

```
1 > source("graphf.R")
2 > graphf(-5,5)
```

the graph in Figure 16.5 is produced.

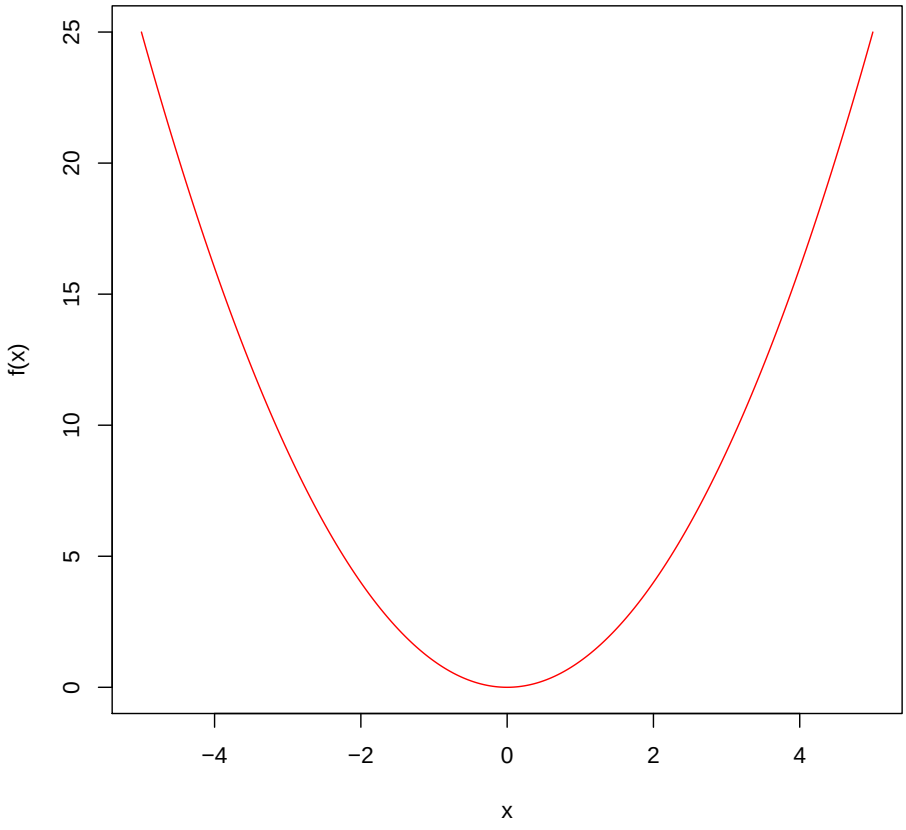


Figure 16.6: Graph of $f(x) = x^2$ drawn using the `graphf` function.

Now, suppose we want to plot a function that is defined piecewise. What must we change in the files `f.R` and `graphf.R`? It turns out, we only have to change the function `f.R`. Now, how do we define a piecewise function in R?

Suppose we want to graph the function

$$f(x) = \begin{cases} x^2 & x \leq 2 \\ 3x + 2 & x > 2 \end{cases} .$$

Notice, that *if* $x \leq 2$ *then* $f(x) = x^2$, and *if* $x > 2$, *then* $f(x) = 3x + 2$. We can use *if statements* when defining `f.R` to correctly describe the piecewise function. See `f.R` below.

`f.R`

```
# Creates the function f(x)
# Input: x (a real number)
# Output: y

f = function(x) {
if (x <= 2) {
   y = x^2
   } else {
   y = 3*x + 2
   }

return(y)
}
```

When run in the Command Window using

`Command Window`

```
> source("graphf.R")
> graphf(-5,5)
```

the following graph in Figure 16.6 is produced.

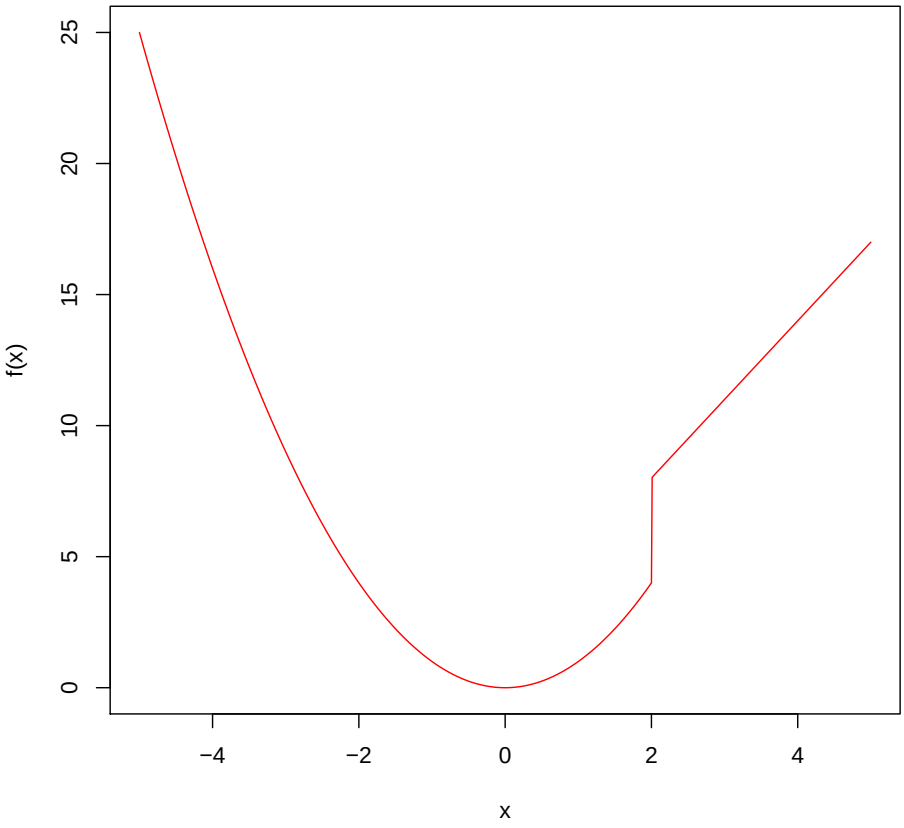


Figure 16.7: Graph of $f(x)$ (from Equation (16.1)) drawn using the `graphf` function.

17.7 R Skills

Biologists have observed a linear relationship between the temperature and the frequency with which a cricket chirps. The following data were measured for the striped ground cricket [53], and has been placed in ascending temperature order.

Temperature °F	Chirps/sec
69.4	15.4
69.7	14.7
71.6	16.0
75.2	15.5
76.3	14.4
79.6	15.0
80.6	16.0
80.6	17.1

Temperature °F	Chirps/sec
82.0	17.1
82.6	17.2
83.3	16.2
83.5	17.0
84.3	18.4
88.6	20.0
93.3	19.8

Let $C(T)$ be the number of chirps per second for temperature T. How do we estimate rates of change for these data? We illustrate using two methods.

Method 1: Using Least-Squares Regression

The first method is to fit a function to the data and use that function to determine average rates of change and estimate instantaneous rates of change. If we plot this data we can see that it seems to have a linear relationship (see Figure 17.2), and thus we can use the methods developed in Chapter 3 to determine the least-squares regression line for the data. The equation for the regression line is

$$C(T) = 0.211925T - 0.309144.$$

The R script used to determine this equation is `CricketChirps.R`.

Now, if we wanted to find the average rate of change in chirps per second as the temperature rose from 70°F to 80°F we could use

$$\frac{C(80) - C(70)}{80 - 70}.$$

This calculation is computed in `CricketChirps.R`. This calculation indicates that as the temperature rises from 70°F to 80°F, there is an average increase of 0.211925 chirps per sec per °F. If we want to estimate the instantaneous rate of change at any point, we can take the derivative of the least-squares regression line $C'(T) = 0.211925$ which implies that the rate of change in chirps per second with respect to temperature is 0.211925 chirps per second per °F.

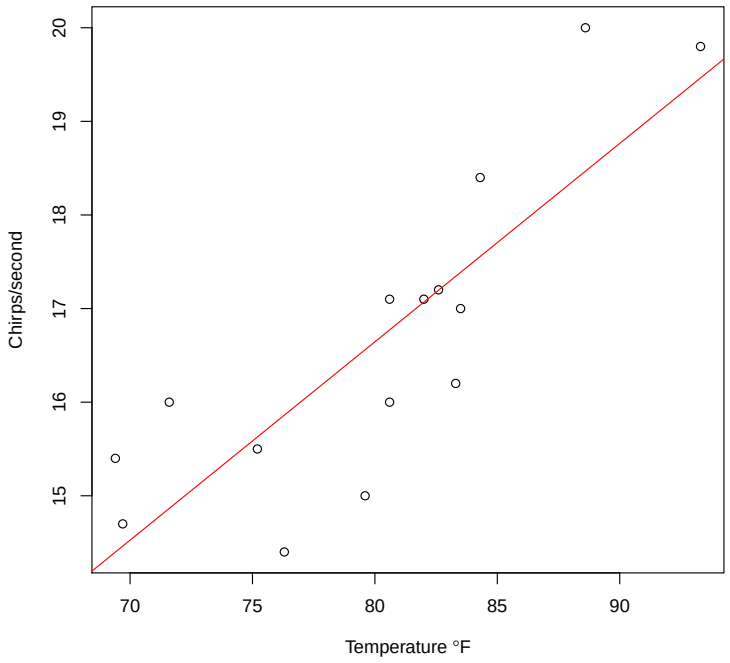


Figure 17.2: Graph of the number of cricket chirps per second with respect to temperature.

CricketChirps.R

```
# Cricket Chirping Data
T = c(69.4, 69.7, 71.6, 75.2, 76.3, 79.6, 80.6, 80.6,
      82.0, 82.6, 83.3, 83.5, 84.3, 88.6, 93.3)
C = c(15.4, 14.7, 16.0, 15.5, 14.4, 15.0, 16.0, 17.1,
      17.1, 17.2, 16.2, 17.0, 18.4, 20.0, 19.8)

# Plot out data
plot(T, C,
   xlab = expression(paste("Temperature ",degree,"F")),
   ylab = "Chirps/second")
# Plot LSR
mod = lm(C~T)
abline(mod,
   col = "red")

# Display the equation
```

```
cat(sprintf("Eqn for LSR: C(T) = %f T + %f",
   coef(mod)[2], coef(mod)[1]), "\n")

# Average rate of change in chirps/sec as temp goes
#    from 70 to 80
f = function(x) {
return(coef(mod)[2] * x + coef(mod)[1])
}

avg = (f(80) - f(70))/(80 - 70)
cat(sprintf("[C(80)-C(70)]/[80-70] = %f chirps/sec/degree F",
   avg), "\n")
```

When the `CricketChirps` R script is run in the Command Window the following output is obtained.

Command Window

```
> source("CricketChirps.R")
Eqn for LSR: C(T) = 0.211814 T + -0.309144
[C(80)-C(70)]/[80-70] = 0.211925 chirps/sec/degree F
```

Method 2: Estimating Directly from the Data

The second method does not require fitting a function to the data. We will use the data directly to estimate the rate of change at each point (except the end points). Thus, if we consider each temperature value T_i and each chirps per second value C_i for $i = 1, 2, \ldots, 15$, we will compute

$$\text{estimated rate of change at } T_i = \frac{1}{2}\left(\frac{C_i - C_{i-1}}{T_i - T_{i-1}} + \frac{C_{i+1} - C_i}{T_{i+1} - T_i}\right)$$

for $i = 2, 3, \ldots, 14$. If we examine our data set, we will see that there are two data points that have the same temperature $(T_7, C_7) = (80.6, 16.0)$ and $(T_8, C_8) = (80.6, 17.1)$. If we use the formula above with this current data set, we will encounter an error when computing

$$\frac{C_8 - C_7}{T_8 - T_7}$$

since $T_8 - T_7 = 0$. How shall we fix this? One solution is to take the average value of chirps per second at $T = 80.6$. Thus, we replace the two data points with the one data point (80.6, 16.6) (notice this change in the R script `CricketChirpsD.R`). The R script `CricketChirpsD.R` with its updated data set, estimates the rate of change at each data point, and then plots these rates of change as well as the original data.

CricketChirpsD.R

```
# Cricket Chirping Data
T = c(69.4, 69.7, 71.6, 75.2, 76.3, 79.6, 80.6,
      82.0, 82.6, 83.3, 83.5, 84.3, 88.6, 93.3)
C = c(15.4, 14.7, 16.0, 15.5, 14.4, 15.0, 16.6,
      17.1, 17.2, 16.2, 17.0, 18.4, 20.0, 19.8)

# Construct an array to store rate of change info
RoC = numeric(length(T)-2)

# Estimate rates of change at each point (not including
#   end points)
for (i in 2:(length(T)-1)) {
   # Average rate of change from i-1 to i
   L = (C[i] - C[i-1]) / (T[i] - T[i-1])

   # Average rate of change from i to i+1
   R = (C[i+1] - C[i]) / (T[i+1] - T[i])

   # Estimate rate of change at i
   RoC[i] = (L+R)/2
   }

# Plot out data
pdf('CricketChirpsD.pdf')
par(mfrow = c(2,1))
plot(T, C,
   col = "blue",
   xlab = expression(paste("Temperature ",degree,"F")),
   ylab = "Chirps/second")

# Plot out rate of change information
plot(T[2:(length(T)-1)], RoC[2:(length(T)-1)],
   type = "b",
   pch = 21,
   col = "green",
   xlab = expression(paste("Temperature ",degree,"F")),
   ylab = expression(paste("Chirps/second/",degree,"F")))
dev.off()
```

When run in the Command Window, `CricketChirpsD` produces the graph shown in Figure 17.3.

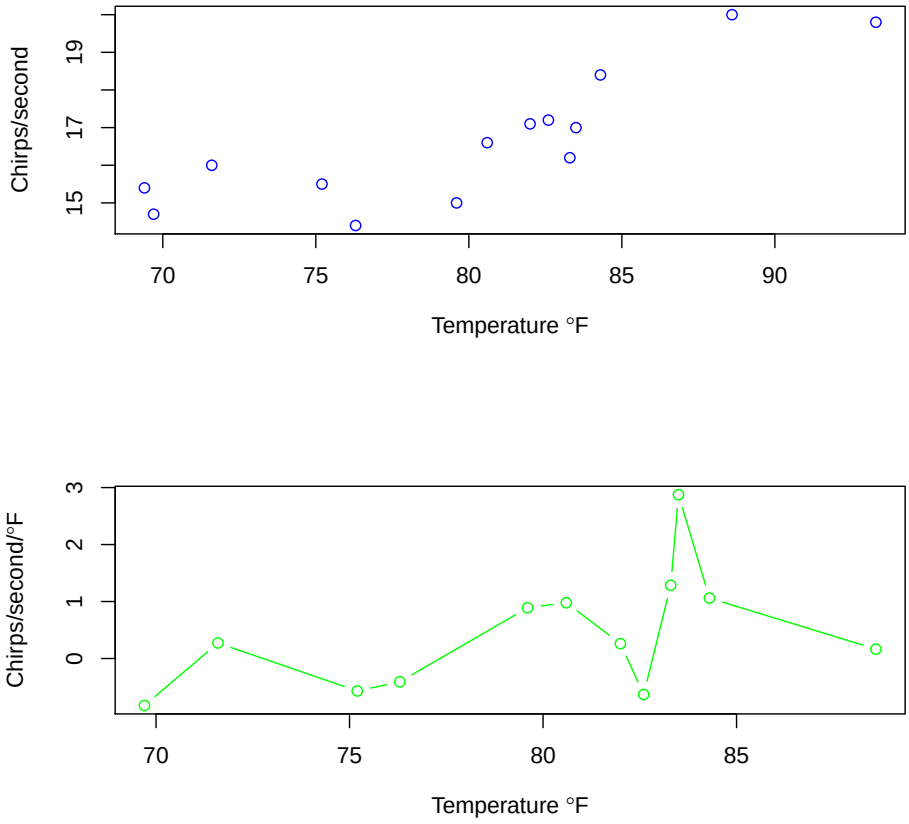


Figure 17.3: The top graph shows the number of cricket chirps per second with respect to temperature. The bottom graph shows the estimated rate of change of cricket chirps per second per ^{circ}F.

18.7 R Skills

In Chapter 15, we used R to numerically approximate limits of functions at a point (see Section 15.3). Here we numerically approximate the derivative of a function at a point.

Recall that the derivative of a function at a point is defined using limits. Thus, the derivative of the function f at the point x is

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

Suppose for example, we want to find the derivative of

$$f(x) = \sqrt[3]{x}$$

at the point $x = 1$. As in Chapter 15, we begin by creating an R script that defines the function f in which we are interested. Recall that $\sqrt[3]{x} = x^{1/3}$.

f.R

```
# Creates the function f(x)
# Input: x (a real number)
# Output: y

f = function(x) {
return(x^(1/3))
}
```

Next, we use the limit definition of a derivative to write a function that approximates the derivative of f at a particular point. Recall from Section 15.3 that we cannot fully evaluate the limit numerically, however we can look at the values the limit approaches from the left and from the right.

derivative.R

```
# Approximates the derivative of the function f
# Make sure that the file f.R is contained in the same folder
#   as this file
source("f.R")

# Input: x = value at which you want to find the derivative
#        h = how close you want to get to the limit (h -> 0)

# Output: LLD = limit of the difference equation from
#               the left side
#         RLD = limit of the difference equation from
#               the right side
```

```
13
14 derivative = function(h,x) {
15
16 # left limit
17 LLD = (f(x+h) - f(x)) / h
18
19 # right limit
20 RLD = (f(x-h) - f(x)) / (-h)
21
22 # Return LLD and RLD
23 return(c(LLD, RLD))
24 }
```

Now, if we want to approximate the derivative $f'(1)$, we would use the following commands in the Command Window,

Command Window

```
1 > source("derivative.R")
2 > derivative(0.1,1)
3 [1] 0.3228012 0.3451062
4 > derivative(0.01,1)
5 [1] 0.3322284 0.3344507
6 > derivative(0.001,1)
7 [1] 0.3332223 0.3334445
8 > derivative(0.0001,1)
9 [1] 0.3333222 0.3333444
```

We see that as we allow $h \to 0$, both the left and right limits approach 0.3333. We might guess that the derivative of f at $x = 1$ is $\frac{1}{3}$. In fact, we know from Example 19.1, that this is the case. It is a good idea to check that the function `derivative.R` works for a function for which we know how to algebraically find the derivative. However, now let us use our newly written function to approximate the derivative of a function for which we do not know how to take the derivative.

Suppose we want to find the derivative of

$$f(x) = \tan(\sin x).$$

Before we apaproximate some derivatives of this function, let us examine a graph of the function to see what we expect. We can utilize the function `graphf.R` which we wrote in Section 16.4. First we modify our `f.R` file.

f.R

```
1 # Creates the function f(x)
```

```
# Input: x (a real number)
# Output: y

f = function(x) {
return(tan(sin(x)))
}
```

Next, to graph the function, we use the `graphf.R` function we wrote. Recall, this function ahs two inputs, `xmin` (the minimum x value to graph) and `xmax` (the maxmimum x value to graph). Since we do not known what the function will look like, let us try graphing from $x = -10$ to $x = 10$. Thus we type the following command in the Command Window,

Command Window

```
> source("graphf.R")
> graphf(-10,10)
```

and the resulting graph is shown in Figure 18.11.

Recall that we formed the limit defintion of the derivative by taking the limit of the slopes of secant lines that passed through $(x, f(x))$ and $(x+h, f(x+h))$. The limit of these secant lines is a line that is tangent to the curve at $(x, f(x))$, and the slope of that tangent line is the derivative. Looking at the graph shown in Figure 18.11 it appears that the slope of the tangent line at $x = 0$ should be positive and have a value of about 1. Let us utilize the `derivative.R` function to see if this is true.

Command Window

```
> source("derivative.R")
> derivative(0.1,0)
[1] 1.001664 1.001664
> derivative(0.01,0)
[1] 1.000017 1.000017
> derivative(0.001,0)
[1] 1 1
```

It definitely appears that $f'(0) = 1$ based on our numerical estimates. Let us "zoom in" on the graph in Figure 18.11 and focus on the portion of the function between $x = 1$ and $x = 2$. This is where one of the peaks occurs. To produce a graph with x values between 1 and 2, we simply use the `graphf.R` function again.

Command Window

```
> source("graphf.R")
> graphf(1,2)
```

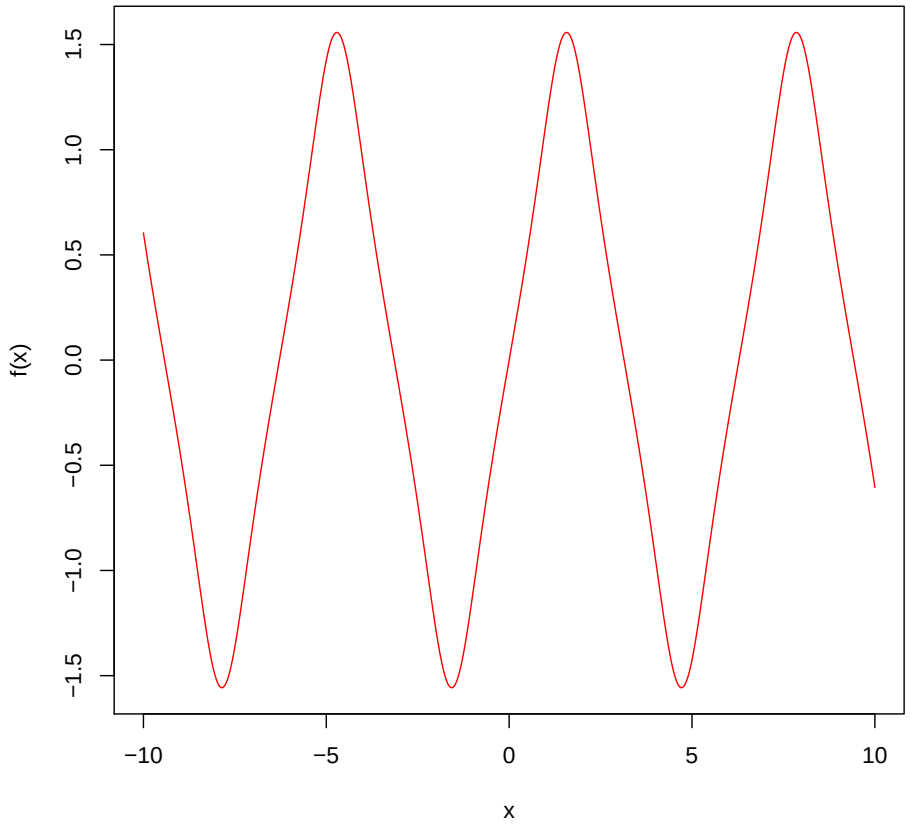


Figure 18.11: Graph of $f(x) = \tan(\sin x)$ for $x \in [-10, 10]$ generated by `graphf.R` function.

The resulting graph is shown in Figure 18.12. It appears that the local maximum of this function occurs arround $x = 1.5$. Note that the tangent line at this local maximum would have a slope of zero. Recall that $\pi/2 = 1.5707\ldots$. If we approximate the derivative at $\pi/2$ we would expect it to be close to zero.

Command Window

```
> source("derivative.R")
> derivative(0.1, pi/2)
[1] -0.1698134  0.1698134
> derivative(0.01, pi/2)
[1] -0.01712612  0.01712612
```

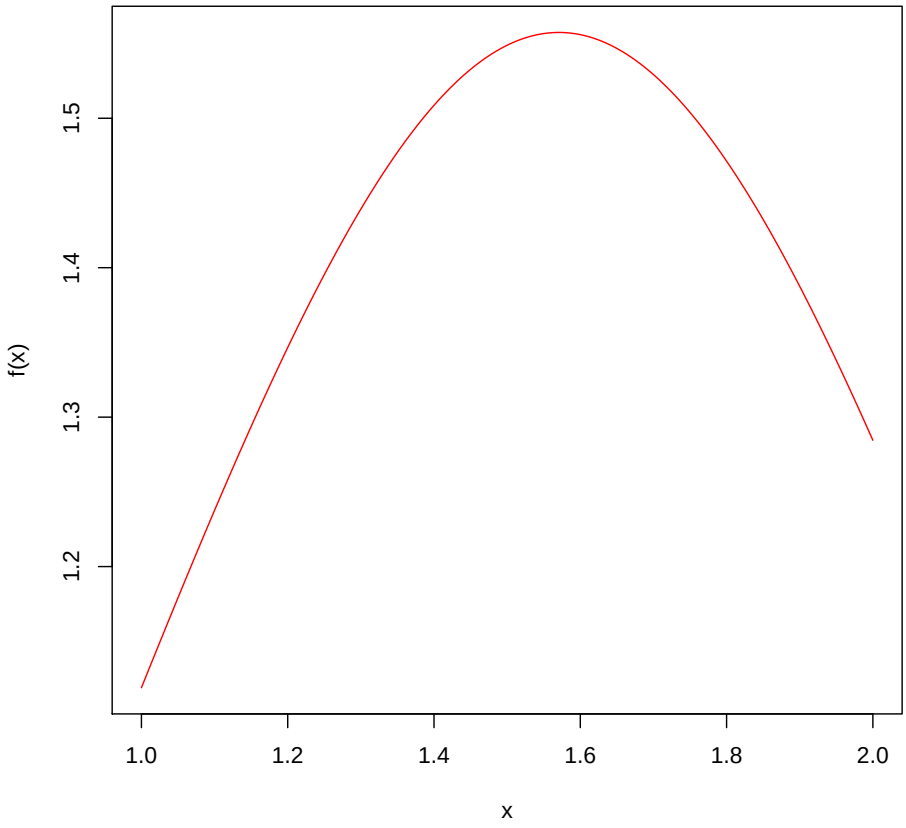


Figure 18.12: Graph of $f(x) = \tan(\sin x)$ for $x \in [1, 2]$ generated by `graphf.R` function.

```
> derivative(0.001, pi/2)
[1] -0.001712758  0.001712758
> derivative(0.0001, pi/2)
[1] -0.0001712759  0.0001712759
```

From our numerical approximations it does appear that $f'(\pi/2) = 0$.

Notice, by using numerical approximations in conjunction with some graphical analysis we were able to predict the value of a derivative at certain points for a function for which we do not yet known how to take the derivative. This is an illustration of the power and importance of utilizing numerical approximations.

20.6 R Skills

There are functions for which we cannot algebraically solve for the local maximum or minimum values. For these functions, we turn to R for assistance.

Consider the function

$$f(x) = \frac{\ln(x)}{x+1}.$$

If we take the derivative of this function we have

$$f'(x) = \frac{1 + \frac{1}{x} - \ln(x)}{(x+1)^2}.$$

The critical points occur where the numerator of $f'(x)$ equals zero and at any points in the domain of $f(x)$ where the function $f'(x)$ is undefined. Notice the domain of $f(x)$ is $x > 0$, so there are no points in the domain of $f(x)$ where $f'(x)$ is undefined. Thus, to find any critical points we set $f'(x) = 0$ which gives us the equation

$$1 + \frac{1}{x} - \ln(x) = 0.$$

How can we solve this equation for x?

It turns out it cannot be done algebraically. However, we can explore the graph of the left hand side of the above equation and find where that graph is equal to 0.

Let us plot the function $g(x) = 1 + \frac{1}{x} - \ln(x)$ over a large portion of its domain, and see where the function crosses the x-axis. To do this, we will use the `plot` command in R. Since we do not know where the function might cross the x-axis, we will start by plotting the function from 0.1 to 100, plotting points at every 0.1 increment.

Command Window

```
> x = seq(0.1, 100, by = 0.01)
> g = function(s) { return(1 + 1/s - log(s)) }
> plot(x, g(x),
+   type = "l",
+   col = "blue",
+   main = "",
+   xlab = "",
+   ylab = "")
> par(new = TRUE)
> abline(h = 0)
```

Notice, when we construct the vector of function values `g` we only need to use the `/` operator to divide by the vector `x`. Additionally, notice that after we plotted just the vectors `x` and `g`, we added to the plot the (horizontal) line $g(x) = 0$ so we could easily see where the function g was passing through the x-axis. The resulting plot is shown in Figure 20.6(a).

Where does $g(x) = 1 + \frac{1}{x} - \ln(x)$ appear to cross the x-axis? Since $g(x)$ crosses the x-axis somewhere between 0 and 10, let us zoom in on that region and see if we can get a better estimate.

Command Window

```
> x = seq(0.1, 10, by = 0.01)
> g = function(s) { return(1 + 1/s - log(s)) }
> plot(x, g(x),
+   type = "l",
+   col = "blue",
+   main = "",
+   xlab = "",
+   ylab = "")
> par(new = TRUE)
> abline(h = 0)
```

The plot produced is shown in Figure 20.6(b).

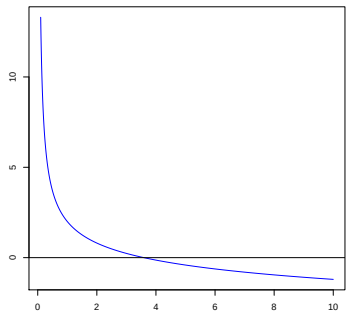
Where does $g(x) = 1 + \frac{1}{x} - \ln(x)$ appear to cross the x-axis now? We can see that $g(x)$ crosses the x-axis somewhere between 3 and 4. Let us zoom in one more time so we can get a better estimate.

Command Window

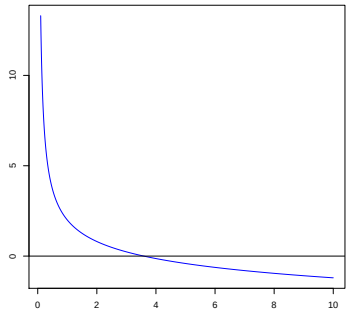
```
> x = seq(3, 4, by = 0.01)
> g = function(s) { return(1 + 1/s - log(s)) }
> plot(x, g(x),
+   type = "l",
+   col = "blue",
+   main = "",
+   xlab = "",
+   ylab = "")
> par(new = TRUE)
> abline(h = 0)
```

The plot produced is shown in Figure 20.6(c).

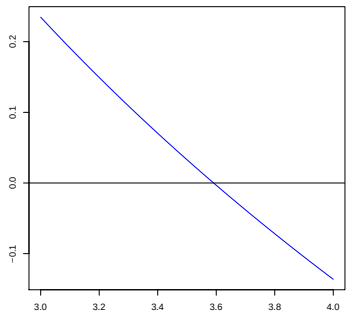
With this last plot, we can estimate that the function $g(x)$ crosses the x-axis at approximately 3.6. We could, of course, continue to zoom in to increase the accuracy of our estimate. However, we will stop here.



(a) $0 \leq x \leq 100$



(b) $0 \leq x \leq 10$



(c) $3 \leq x \leq 4$

Figure 20.6: Plot of $g(x) = 1 + \frac{1}{x} - \ln(x)$.

21.4 R Skills

Example 21.2 used a total of 10 rectangles to estimate the area bounded between the curve $f(x) = 1 - x^2$ and the horizontal axis. If greater accuracy is desired we would need to use smaller intervals and a greater number of rectangles. As the number of rectangles increases, the calculations of the total area quickly become tedious to do by hand, so it is helpful to do these calculations in R.

To estimate the area bounded between the curve $f(x) = 1-x^2$ and the horizontal axis using 20 rectangles, start by editing the function `f.R`, which we wrote and used in Sections 15.3, 16.4, and 18.7.

f.R

```
# Creates the function f(x)
# Input: x (a real number)
# Output: y

f = function(x) {
return(1 - x^2)
}
```

Next, write a function in R that will compute the total area by summing the areas of all the rectangles. To do this, we can use a loop. We first write a function that utilizes the left endpoint of each subinterval.

Lrect.R

```
# Approximates the area under the function f over a
#   given interval
# Make sure the file f.R is contained in the same folder
#   as this file
source("f.R")

# Input: xmin = left end of interval
#        xmax = right end of interval
#        n = how many rectangles to use

# Output: sum = approx of the area under f over a
#               given interval

Lrect = function(xmin, xmax, n) {

# Initially set sum to zero
sum = 0

# Calculate the width of the rectangles
```

```
deltax = (xmax - xmin) / n

# One pass through the for loop for each rectangle
for (i in 0:(n-1)) {
   # Add onto the current sum the area of the current triangle
   sum = sum + f(xmin + i*deltax) * deltax
   }

return(sum)
}
```

We can check that this function works correctly (something which is always a good idea when you are coding something new) by testing it using the case $n = 10$ since we already know the answer from Example 21.2, then we can analyze how our answer changes using $n = 20,\ 50,\ 100$.

Command Window

```
> A = Lrect(-1, 1, 10)
> A
[1] 1.32
> A = Lrect(-1, 1, 20)
> A
[1] 1.33
> A = Lrect(-1, 1, 50)
> A
[1] 1.3328
> A = Lrect(-1, 1, 100)
> A
[1] 1.3332
```

First, for $n = 10$, the total area of the rectangles is 1.32 units^2 which is the same answer obtained in Example 21.2. Next, as the number of rectangles used increases, i.e. increasing the value of n, the total area approaches 1.3333... units^2 or $\frac{4}{3}$ units^2.

Next, consider a function similar to `Lrect.R` that uses the right endpoint of each subinterval.

Rrect.R

```
# Approximates the area under the function f over a
#   given interval
# Make sure the file f.R is contained in the same folder
#   as this file
source("f.R")

```

```
# Input: xmin = left end of interval
#        xmax = right end of interval
#        n = how many rectangles to use

# Output: sum = approx of the area under f over a
#               given interval

Rrect = function(xmin, xmax, n) {

# Initially set sum to zero
sum = 0

# Calculate the width of the rectangles
deltax = (xmax - xmin) / n

# One pass through the for loop for each rectangle
for (i in 1:n) {
   # Add onto the current sum the area of the current triangle
   sum = sum + f(xmin + i*deltax) * deltax
   }

return(sum)
}
```

Notice the only difference between `Lrect.R` and `Rrect.R` is on line 23 of the code where we start the for loop. For the left endpoints of the subintervals, we use `i in 0:(n-1)`, whereas for the right endpoints of the subintervals we use `i in 1:n`.

We can additionally write the code for a method which uses trapezoids instead of rectangles.

Trap.R

```
# Approximates the area under the function f over
#   a given interval
# Make sure the file f.R is contained in the same
#   folder as this file
source("f.R")

# Input: xmin = left end of interval
#        xmax = right end of interval
#        n = how many trapezoids to use

# Output: sum = approx of the area under f over a
#               given interval

```

```
Trap = function(xmin, xmax, n) {

# Initially set sum to zero
sum = 0

# Calculate the width of the rectangles
deltax = (xmax - xmin) / n

# One pass through the for loop for each trapezoid
for (i in 0:(n-1)) {
   # Add onto the current sum the area of the current
   # trapezoid
   sum = sum +
         (f(xmin + i*deltax) +
         f(xmin + (i+1)*deltax))/2 * deltax
   }

return(sum)
}
```

Example 21.7 gave estimates for the exact area A bounded between $f(x) = x^2$ and the horizontal axis over the interval [0,1] as

$$0.285 < A < 0.410.$$

The estimate was obtained by first using the left endpoints of the subintervals with $n = 10$ and then using the right endpoints of the subintervals with $n = 10$. Now use the functions `Lrect.R`, `Rrect.R`, and `Trap.R` to obtain a better estimate.

Command Window

```
> Lrect(0, 1, 10)
[1] 0.285
> Rrect(0, 1, 10)
[1] 0.385
> Trap(0, 1, 10)
[1] 0.335
> Lrect(0, 1, 50)
[1] 0.3234
> Rrect(0, 1, 50)
[1] 0.3434
> Trap(0, 1, 50)
[1] 0.3334
> Lrect(0, 1, 100)
[1] 0.3284
```

```
> Rrect(0, 1, 100)
[1] 0.3384
> Trap(0, 1, 100)
[1] 0.3333
```

For $n = 10$, the estimate was

$$0.285 < A < 0.410,$$

and from the trapezoid method,

$$A \approx 0.3350.$$

For $n = 50$, the estimate was

$$0.3234 < A < 0.3434,$$

and from the trapezoid method,

$$A \approx 0.3334.$$

For $n = 100$, the estimate was

$$0.3284 < A < 0.3384,$$

and from the trapezoid method,

$$A \approx 0.3333.$$

It would appear that the estimates using the rectangles with left and right endpoints indicate that the exact area is $\frac{1}{3}$. This is supported by the estimates using the trapezoid method which also shows the estimated area is approaching $\frac{1}{3}$. Thus, we might hypothesize that the exact area bounded between $f(x) = x^2$ and the horizontal axis over the interval [0,1] is $A = \frac{1}{3}$. Using the techniques in the next chapter, we will be able to show that this is correct.

22.6 R Skills

As we saw in Example 22.12 we can approximate a definite integral using areas of rectangles bounded between the integrand function and the horizontal axis over the interval defined by the limits of integration. We could have also used trapezoids. In Section 21.4 we developed a set of R functions to help us approximate the area under a curve. Since those R functions do not depend on the curve being non-negative, we can use those R functions to approximate the values of definite integrals.

Let us use the functions `f.R`, `Lrect.R`, `Rrect.R`, and `Trap.R` to estimate the area bound between the functions $f(x) = 1 - x^2$ and the horizontal axis over the interval [0,2]. If we use four intervals, i.e. $n = 4$ we should get the same answer we found in Example 22.12. As we increase the number of intervals, we approach the exact value of the integral,

$$\int_0^2 \left(1 - x^2\right) dx.$$

First, we must modify the function `f.R` appropriately.

`f.R`

```
# Creates the function f(x)
# Input: x (a real number)
# Output: y

f = function(x) {
return(1 - x^2)
}
```

Next, using `Lrect.R`, let us estimate the value of the integral using $n = 4$, 10, 25, 100, 500, and 1000.

Command Window

```
> Lrect(0, 2, 4)
[1] 0.25
> Lrect(0, 2, 10)
[1] -0.28
> Lrect(0, 2, 25)
[1] -0.5088
> Lrect(0, 2, 100)
[1] -0.6268
> Lrect(0, 2, 500)
[1] -0.6587
> Lrect(0, 2, 1000)
[1] -0.6627
```

Notice, the initial approximation, using $n = 4$, produces a very different estimation than when we increase the value of n. Now, using `Rrect.R`, let us estimate the value of the integral using $n = 4$, 10, 25, 100, 500, and 1000. Do you expect these approximations to be greater or smaller than the estimates made using the left endpoints of the intervals?

Command Window

```
> Rrect(0, 2, 4)
[1] -1.75
> Rrect(0, 2, 10)
[1] -1.08
> Rrect(0, 2, 25)
[1] -0.8288
> Rrect(0, 2, 100)
[1] -0.7068
> Rrect(0, 2, 500)
[1] -0.6747
> Rrect(0, 2, 1000)
[1] -0.6707
```

Let $A = \int_0^2 (1 - x^2)dx$, and thus A is the exact value of the integral. Using the estimates using `Lrect.R` and `Rrect.R`, when $n = 4$ we can estimate

$$-1.7 < A < 0.25,$$

when $n = 10$ we can estimate

$$-1.08 < A < -0.28,$$

when $n = 25$ we can estimate

$$-0.8288 < A < -0.5088,$$

when $n = 10$ we can estimate

$$-0.7068 < A < -0.6268,$$

when $n = 10$ we can estimate

$$-0.6747 < A < -0.6587$$

when $n = 10$ we can estimate

$$-0.6707 < A < -0.6627$$

Notice, as n increases, the interval we use to estimate A becomes smaller and smaller. Thus, we see we are approaching the exact value of A.

Lastly, let us use `Trap.R` to estimate the value of the integral using $n = 4$, 10, 25, 100, 500, and 1000. How do you expect these estimates to compare to the estimates made using `Lrect.R` and `Rrect.R`?

Command Window

```
> Trap(0, 2, 4)
[1] -0.75
> Trap(0, 2, 10)
[1] -0.68
> Trap(0, 2, 25)
[1] -0.6688
> Trap(0, 2, 100)
[1] -0.6668
> Trap(0, 2, 500)
[1] -0.6667
> Trap(0, 2, 1000)
[1] -0.6667
```

What value do the estimates seem to be approaching? It appears that as we increase the size of n, the estimates approach the value -0.6667. In fact, if we allowed R to show us a great number of digits in the calculation, we would see that the estimates approach the value -0.666666... $= -\frac{2}{3}$. If we solve $A = \int_0^2 (1 - x^2)dx$ algebraically, we would find that, indeed, $A = -\frac{2}{3}$.

25.4 R Skills

Waiting Times: Birders and Bad Luck

Birding is one of the most popular outdoor activities in the world and many birders keep a "life list" of species of birds they have observed. It is very common for the local "birder network" to inform each other when a particularly unusual or rare species is in an area. Suppose you wanted to add the broad-winged hawk (*Buteo platypterus*) to your life list, and someone told you that two of these hawks were observed in an hour at a nearby vantage point (perhaps during their migration south to Florida for the winter). If you then rushed over to the viewing site, how long do you suppose you would have to wait to see a hawk? Since it appears the mean time between bird sightings is 30 minutes (there were two birds seen in an hour), and you might think of yourself as randomly arriving at the vantage point between two bird sightings, you would expect that on average you'd have to wait about 15 minutes to see a hawk and add that species to your life list. This is thinking of yourself as on-average arriving in the middle of two sightings of hawks. We will see below whether this intuition is correct, but first we will calculate the probability of seeing a hawk in some time period.

If we assume an exponential distribution of times of arrivals for hawks passing this vantage point, what is the probability you will see at least one hawk in 30 minutes? If we let T be a random variable that has the exponential distribution with mean waiting time of $\mu = 0.5$ where time is measured in hours, then the probability a waiting time is less than 30 minutes is

$$P(T \leq 0.5) = \int_0^{0.5} \frac{1}{0.5} e^{-t/0.5} dt = e^{-t/0.5}\Big|_0^{0.5} = 1 - e^{-1} = 0.632$$

This would seem to imply that you would have a reasonable chance (63%) of seeing at least one hawk if you spent 30 minutes at the vantage point.

Now let us determine if your intuition is correct about the mean time you would need to wait to see a hawk. We will use R to randomly generate 10 waiting times. Essentially, we will "randomly sample" the exponential probability distribution 10 times.

The exponential probability distribution with mean waiting time 30 minutes (or 0.5 hours), is

$$P(T \leq t) = \int_0^t \frac{1}{0.5} e^{-u/0.5} du = 1 - e^{-2t}.$$

Let p be the probability that your waiting time T is less than or equal to t; then

$$p = 1 - e^{-2t} \;\Rightarrow\; e^{-2t} = 1 - p \;\Rightarrow\; t = -0.5 \ln(1 - p).$$

Since p is a probability, it has a value between 0 and 1. In R we can use the built-in function `runif` to generate a random number between 0 and 1 (see R Skill Section 10.5). Using `runif(10)` we can generate 10 random numbers between 0 and 1. In the Command Window, we can generate 10 waiting times from the exponential probability distribution.

Command Window

```
> t = -0.5 * log(1 - runif(10))
> t
[1] 1.05927365 0.03279604 0.08926527 0.69024280 0.30148882
[6] 0.78500964 0.09135814 0.20106842 0.12306942 0.12813980
```

This is one way to simulate a random variable if you can both find the distribution function and take its inverse (which is what we have done by solving for t in terms of p). The mean time between arrivals for the sample of waiting times is obtained by simply taking their arithmetic mean, `mean(t)`, to get 0.3502 hours or about 21 minutes.

This particular set of times between arrivals for a hawk includes some times that are very short compared to the mean of 0.5 hour and some that are very long compared to it. Note that the values in the array `t` are in hours. We can use R to convert those values to hours and minutes using the `floor` and `%%` functions. For a real number a, the command `floor(a)` finds the largest integer less than or equal to a. For real numbers a and b, the command `a%%b` finds the remainder of a/b. Typically, we use integer values for b. Once we have the array `t`, we can convert those values to hours and minutes in the Command Window.

Command Window

```
> t = -0.5 * log(1 - runif(10))
> minutes = round((t*60)%%60)
> W = data.frame(t, minutes)
> W
            t minutes
1  1.13435802       8
2  0.19836296      12
3  0.48067757      29
4  0.79236067      48
5  0.20152156      12
6  0.73155512      44
7  1.10057541       6
8  0.04531145       3
9  0.20964524      13
10 1.14661859       9
```

The command `floor(t)` determines the number of hours for each waiting time. Note the floor of any value in the array `t` with a value less than one will be zero.

The command `(t*60)%%60` multiplies each value in the array `t` by 60 (converting it to minutes), then divides that number by 60 and finds the remainder. Note, we round the resulting value to the nearest minute (the nearest whole number).

Suppose someone was at the vantage point, stayed there all day, saw the first hawk at 5:00 AM, and observed 10 more hawks such that the time between sightings (in fractions of an hour) is given by the sampled values in `t`. Then the list of times at which hawks were observed for this particular set of interarrival times is

05:00, 05:58, 07:00, 07:09, 07:16, 07:41, 08:12, 08:28, 08:35, 10:04, 10:06.

We can use R to calculate these values as well. For this calculation we will use the R function `cumsum` which calculates the cumulative sum of an array. For example, if we want to find the cumulative sum of the array

$$c(0, 1, 2, 3, 4, 5),$$

then the function `cumsum` will produce the array

$$c(0, 1, 3, 6, 10, 15)$$

where each value x_i in the latter array is the sum of the first i terms in the original array. In the Command Window, we compute the time at which each of the 10 hawks were sighted.

Command Window

```
> S = cumsum(t) + 5
> hours = floor(S)
> minutes = round((S*60)%%60)
> W = data.frame(hours, minutes)
> W
   hours minutes
1      6       8
2      6      20
3      6      49
4      7      36
5      7      48
6      8      32
7      9      38
8      9      41
9      9      54
10    11       2
```

Note that we add 5 to the cumulative sum since the initial hawk sighting occured at 5:00 AM. After we calculate the time in hours of each sighting, we

again converted those values to hours and minutes to obtain the lists of times of the sightings. Note that this type of conversion would compute the times in military time, so any time occurring after 12:00 PM would have hour values greater than 12.

The first sighting occurred at 5:00 AM and the 11^{th} sighting occurred at 11:02 AM. Imagine a birder going to the vantage point at sometime during this period. Lacking additional information we will assume that the birder arrives at some "random" time between 05:00 and 11:02. The definition of "random" here is that it is equally likely that the birder will arrive at any time in this time period. When it is equally likely that any particular value is selected from an interval, we use the probability distribution function the *uniform distribution*. Recall the density function for a uniform random variable on the interval from a to b is:

$$f(t) = \begin{cases} 0, & \text{if } t < a \\ 1/(b-a), & \text{if } a < t \leq b \\ 0, & \text{if } t > b. \end{cases}$$

We have already seen how to sample from a uniform probability distribution in R Skills Section 10.5, though we did not call it a uniform probability distribution at that point. To use `runif` to sample from a uniform disitrubtion over $[a, b]$, we use the command

```
runif(1)*(b-a)+a
```

We will use R to simulate 10 birders arriving at random times between 05:00 AM and 11:02 AM. Recall we saved the time values in the array `S`. Thus, we want to sample from the uniform distribution over [5, max(S)]. In the Command Window, we sample 10 values from the uniform distribution and convert those values into time of day.

Command Window

```
> R = runif(10) * (max(S) - 5) + 5
> R
[1]  5.669988  5.200347  8.049822 10.303035  9.494667
[6]  5.409801  6.897304  7.670676  5.759993  8.121796
> hours = floor(R)
> minutes = round((R*60)%%60)
> data.frame(hours, minutes)
   hours minutes
1      5      40
2      5      12
3      8       3
4     10      18
5      9      30
```

```
6       5       25
7       6       54
8       7       40
9       5       46
10      8        7
```

Thus, the times at which the birders arrive are

5:40, 5:12, 8:03, 10:18, 9:30, 5:25, 6:54, 7:40, 5:46, and 8:07.

How long does each birder have to wait to see a hawk? Take for example, the birder who shows up at 5:40 AM. The next hawk passes the vantage point at 5:46. So this birder will only have to wait 6 minutes. However, for the birder who shows up at the vantage point at 9:30, the next hawk will not pass by until 10:18. This birder will have to wait 48 minutes. Let us use R to compute the time each birder has to wait to see the next hawk, and find the average time a randomly arriving birder will have to wait for a sighting.

The array `S` contains the times at which each hawk passes the vantage point. In R the command `S[S>R[1]]` finds all the values in `S` which are greater than `R[1]` (the time the first sample birder arrives). Thus, the command returns the times of all the sightings the birder who arrived at time `R[1]` would be able to see. If we want the time of the first such sighting, we can take the minimum of the resulting array. Thus, `min(S[S>R[1]])` is the minimum value of `S` which is greater than `R[1]`. If we then subtract the time at which the birder arrived, we have calculated how long the birder had to wait for her first sighting. We can construct a for loop to find the waiting time for each birder (looping through the length of `R`).

Command Window

```
>
> for (i in 1:length(R)) {
+ D[i] = min(S[S>R[i]]) - R[i]
+ }
> hours = floor(D)
> minutes = round((D*60)%%60)
> data.frame(hours, minutes)
   hours minutes
1      0      28
2      0      56
3      0      29
4      0      44
5      0       9
6      0      43
7      0      43
8      0       8
```

```
9      0      22
10     0      25
> meanD = mean(D*60)
> meanD
[1] 30.78667
```

After we computed the waiting times, we converted those values to hours and minutes. It is interesting to observe that while some birders had to wait fewer than 10 minutes for a sighting, others had to wait for nearly an hour. When we compute the mean waiting time of the 10 birders (in minutes) we found that the average time a birder had to wait for a sighting was roughly 30 minutes.

Recall, we hypothesized that a randomly arriving birder would have to wait on average 15 minutes. However, our random simulation shows the average waiting time to be closer to double this amount of time. Why is this? It is possible of course that this is an artifact of the particular arrival times of hawks and birders we obtained. To test this idea, we could try rerunning the "experiment" of simulating bird and birder arrival times and computing the average waiting time of each birder. The R script file `HawkWaitingTimes.R` simulates the appearance of 10 hawks at the vantage point after 5:00 AM, along with the random arrival of 10 birders and the amount of time they have to wait to see the next hawk. Run the `HawkWaitingTimes.R` script file multiple times. What are the average birder waiting times?

HawkWaitingTimes.R

```
# Hawk Waiting Times

# Generate a sample of 10 waiting times
T = -0.5 * log(1 - runif(10))

# Calculate times of sightings
S = 5 + cumsum(T)

# Display waiting times in terms of hours and minutes
cat("\nWaiting Times  //  Sighting Times\n")
cat("               //   5:00  \n")
for (i in 1:length(T)) {
   cat(sprintf("%2.f h  %2.f min", floor(T[i]),
       round((T[i]*60)%%60)))
   cat(sprintf("   //  "))
   cat(sprintf("%2.f:%02.f", floor(S[i]),
       round((S[i]*60)%%60)), "\n")
   }

# Calculate and print mean waiting time in minutes
meanT = mean(T*60)
```

```
cat(sprintf("\nMean waiting time is %4.1f minutes\n\n",
    meanT))

# Generate a random time at which the birder shows up
R = runif(10)*(S[length(S)]-5)+5

# Generate an array to contain time of next sighting
D = numeric(length(R))

# For each birder, find the time of the next hawk
#   sighting
for (i in 1:length(R)) {
   v = min(S[S>R[i]])  #time of next sighting
   D[i] = v - R[i]     #time of next sighting - time of
                       #birder arrival
   }

# Display arrival and waiting times
cat("10 Birders randomly show up\n")
cat("\nArrival Times  //  Waiting Times\n")
for (i in 1:length(T)) {
   cat(sprintf("   %02.f:%02.f", floor(R[i]),
       round((R[i]*60)%%60)))
   cat("       //    ")
   cat(sprintf("%3.f min\n", D[i]*60))
   }

# Calculate and print mean waiting time in minutes of
# the 10 birders
meanD = mean(D*60)
cat("\nMean waiting time of 10 random birders")
cat(sprintf(" is %4.1f minutes\n\n", meanD))
```

When we run the `HawkWaitingTimes.R` script two times, we generate the following ouput in the Command Window.

Command Window

```
> source("HawkWaitingTimes.R")

Waiting Times  //  Sighting Times
               //   5:00
 0 h  27 min   //   5:27
 0 h   7 min   //   5:35
 0 h  34 min   //   6:08
 0 h   3 min   //   6:11
```

```
 0 h   4 min   //   6:15
 1 h   8 min   //   7:23
 0 h  17 min   //   7:40
 0 h  12 min   //   7:52
 0 h   6 min   //   7:59
 1 h  17 min   //   9:15

Mean waiting time is 25.5 minutes

10 Birders randomly show up

Arrival Times  //  Waiting Times
   05:40       //     29 min
   06:12       //      3 min
   05:49       //     19 min
   05:50       //     19 min
   07:42       //     10 min
   08:16       //     60 min
   06:23       //     60 min
   07:52       //      0 min
   06:57       //     26 min
   08:29       //     47 min

Mean waiting time of 10 random birders is 27.2 minutes

> source("HawkWaitingTimes.R")

Waiting Times  //  Sighting Times
               //   5:00
 0 h  11 min   //   5:11
 0 h  39 min   //   5:50
 0 h  57 min   //   6:47
 0 h  19 min   //   7:06
 0 h  24 min   //   7:30
 0 h   5 min   //   7:35
 1 h  48 min   //   9:23
 0 h  35 min   //   9:58
 0 h  14 min   //  10:12
 0 h   8 min   //  10:19

Mean waiting time is 31.9 minutes

10 Birders randomly show up

Arrival Times  //  Waiting Times
   06:06       //     41 min
```

```
   08:35       //     47 min
   07:43       //    100 min
   08:34       //     49 min
   05:52       //     55 min
   09:26       //     32 min
   10:11       //      0 min
   09:23       //     35 min
   06:42       //      5 min
   10:05       //      6 min

Mean waiting time of 10 random birders is 37.0 minutes
```

If you run the `HawkWaitingTimes.R` script file many times you will observe that the mean waiting time of the birders is very rarely as low as 15 minutes. The reason for this is a bit subtle. You can visualize what happens by thinking of a timeline starting at 05:00, ending at 11:02 and marking on it the arrival times of the hawks. Now imagine tossing a dart randomly at this timeline. Where is the dart most likely to hit? It is much more likely that it will land in a time segment that has a long interarrival time (such as the time between 9:30 to 10:18) than in one of the shorter interarrival times (such as the one from 5:12 to 5:25). Thus, it is more likely that a randomly arriving birder will have a longer wait than the wait time we intuitively thought of as one-half of the average waiting time between birds that the same birder would observe if he or she stayed at the vantage point all day. This is commonly called "bad luck." The official name for this is "length-biased sampling," which means that you are more likely to arrive in a longer interarrival time than in a shorter one.

26.2 R Skills

The solutions to differential equations are continuous functions. However, there are methods to approximate values of the solution function over a series of values for the independent variable (usually time). For example, we could approximate the solution to the differential equation $\frac{dy}{dx} = xy$ by approximating the solution $y(x)$ at x-values 0, 1, 2, 3, 4, ...

In this text, we do not introduce these methods for numerically approximating the solutions to differential equations. Some of these methods would be discussed in detail in a differential equations course. See Brannan and Boyce's Differential Equation textbook [9] or Burden and Faires' Numerical Analysis textbook [10] for an explanation of the Euler Method (the simplest numerical method for approximating the solution to a differential equation).

For numerical approximations of solutions to differential equations in this course, we turn to R. Various packages that we can load in R contain functions to numerically "solve" differential equations, and we discuss how to use these ODE "solvers" here. An R script to solve an ODE, using the `deSolve` package, is shown in `odef.R`.

`odef.R`

```
# Load the package:
require(deSolve)

# Declare your parameters if there are any
a = 1
parms = c(a)

# Declare initial conditions
x_0 = 0
y_0 = 1
start = c(x_0, y_0)

# Write the model
odef = function(t, n, parms) {

x = n[1]
y = n[2]

with(as.list(parms),
{
   # Write the differential equations
   dx = 1
   dy = a*x*y

```

```
25    res = c(dx, dy)
26
27    list(res)
28    })
29 }
30
31 # Declare times
32 times = seq(x_0, 1, by = 0.01)
33
34 # Solve the model using rk4 (a 4th-order Runge-Kutta method)
35 output = as.data.frame(rk4(start, times, odef, parms))
36
37 # Take a look at the names of the labels that rk4
38 # automatically creates
39 names(output)
40
41 # We want to plot the second column, "2"
42 plot(output$"2",
43    type = "l",
44    col = "blue",
45    ylab = "y(x)",
46    xlab = "x",
47    ylim = c(0, 3.5),
48    xaxt = "n")
49 axis(1,                          # The horizontal axis
50    at = seq(0, 100, by = 20),    # These are the current labels
51    labels = seq(0, 1, by = 0.2)) # Change them to these
```

Notice that the above differential equation we are asked to solve, $\frac{dy}{dx} = xy$, depends on x. We can still solve this with our solver, however we need to treat both y and x as functions of another variable, t. Hence, $x = x(t)$ and $y = y(x(t))$. In line 22, by setting `dx = 1`, we are essentially treating $x(t)$ and t as the same, but in a manner in which R can handle them.

The output from the `rk4` function gives us a data array containing our desired solutions. The first column is reserved for `n[1]`, which we set in line 16 of the code to be x, and the second column contains `n[2]`, or our y values. Observe that the first column is identical to the values stored in `times`, for the reasons mentioned above.

Finally, notice that the plot contains the corresponding index of our computed y values. Intuitively, this makes sense, since our y values are stored as an array. However, we would like to fix this so that the axis reflects the values of x that $y(x)$ is taking on. We do this in lines 48 - 50 of the code, using the `axis` command. The plot is shown in Figure 26.1.

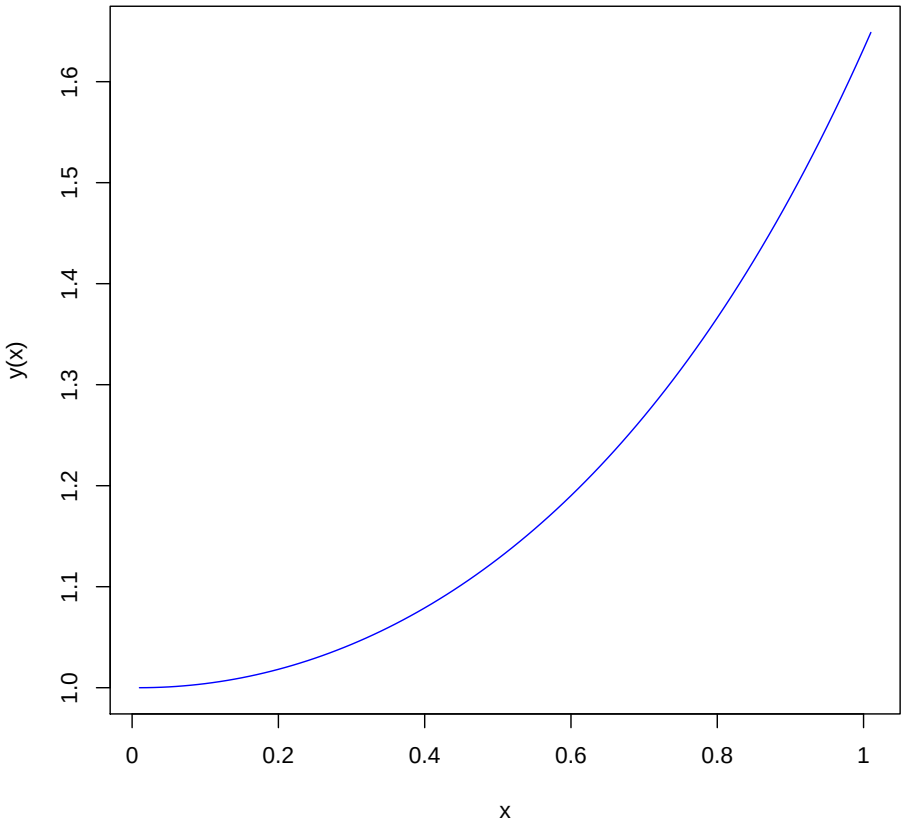


Figure 26.1: Graph produced using `odef.R`, set to solve $dy/dx = xy$.

If we want to explore a variety of different initial conditions, we can construct an R script with a for loop, where each pass through the loop numerically approximates the solution using a different initial condition. Running the `ODESolutions.R` script will produce the graph shown in Figure 26.2.

`ODESolutions.R`

```
# Make sure that your first graph is still open,
# and that the original graph was drawn so that
# the y-axis goes from 0 to 3.5
y_vals = c(0, 3.5)

# Vector of initial conditions
```

```
y_0 = seq(0.2, 2, by = 0.2)

# For-loop
for (i in 1:length(y0)) {
   # Re-set start each time
   start = c(x_0, y_0[i])

   # Solve the ODE
   output = as.data.frame(rk4(start, times, odef, parms))

   # Plot our values on the same graph
   par(new = TRUE)
   plot(output$"2",
      type = "l",
      col  = c(i),
      ylab = "",
      xlab = "",
      ylim = y_vals,
      xaxt = "n",
      yaxt = "n")
   }
```

Now, suppose we are told that the rate of change of a population is given by the equation

$$\frac{dy}{dt} = 0.25y(t) \ln\left(\frac{1000}{y(t)}\right)$$

and we want to know how the population changes over 20 years if the population starts with 50, 100, 500, 1000, and 1500 individuals. We can create a new R script to do this for us, based off of the codes contained in `odef.R` and `ODESolutions.R`.

`ode_pop.R`

```
# Load the package:
require(deSolve)

# Declare your parameters if there are any
a = 0.25
parms = c(a)

# Declare initial conditions
y_0 = 50
start = c(y_0)

# Write the model
```

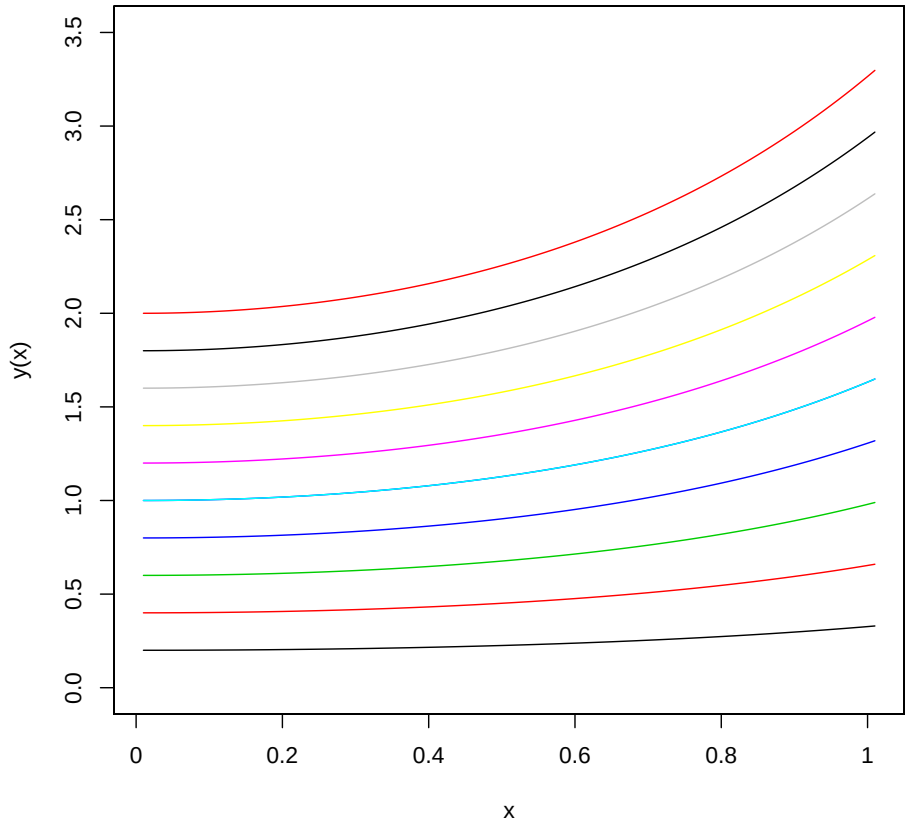


Figure 26.2: Graph produced using `ODESolutions.R` set to solve $dy/dx = xy$ with various initial conditions.

```
13 odef = function(t, n, parms) {
14
15 y = n[1]
16
17 with(as.list(parms),
18 {
19    # Write the differential equations
20    dy = a * y * log(1000/y)
21
22    res = c(dy)
23
```

```
   list(res)
   })
}

# Declare times
times = seq(0, 20, by = 0.01)

# Solve the model using rk4 (a 4th-order Runge-Kutta method)
output = as.data.frame(rk4(start, times, odef, parms))

# Prepare to add in a legend
par(mar = c(5.1, 4.1, 4.1, 11.1), xpd = TRUE)

# Take a look at the names of the labels that rk4
# automatically creates
names(output)

# We want to plot the first column, "1"
plot(output$"1",
   type = "l",
   col  = c(1),
   ylim = c(0, 1500),
   ylab = "y(t)",
   xlab = "t",
   xaxt = "n")
axis(1,                          # The horizontal axis
   at = seq(0, 2000, by = 500), # These are the current labels
   labels = seq(0, 20, by = 5)) # Change them to these

# Vector of initial conditions
y_0 = c(100, 500, 1000, 1500)

# For-loop
for (i in 1:length(y0)) {
   # Re-set start each time
   start = c(y_0[i])

   # Solve the ODE
   output = as.data.frame(rk4(start, times, odef, parms))

   # Plot our values on the same graph
   par(new = TRUE)
   plot(output$"1",
      type = "l",
      col  = c(i+1),
      ylim = c(0, 1500),
```

```
70        ylab = "",
71        xlab = "",
72        xaxt = "n",
73        yaxt = "n")
74     }
75
76  legend("right",
77     inset = c(-0.41, 0),
78     lty = c(1, 1, 1, 1, 1),
79     c("50", "100", "500", "1000", "1500"),
80     col = c(1, 2, 3, 4, 5))
```

Notice there were some changes to this file. First, since we were solving a differential equation which did not depend explicitly on our dependent variable, we were able to abandon the trick of adding a term like the `dx = 1` from before. Second, we changed `times` in line 29 so that solutions are shown for time values from 0 to 20. Now, if we run the R script `ode_pop.R` we produce the graph shown in Figure 26.3.

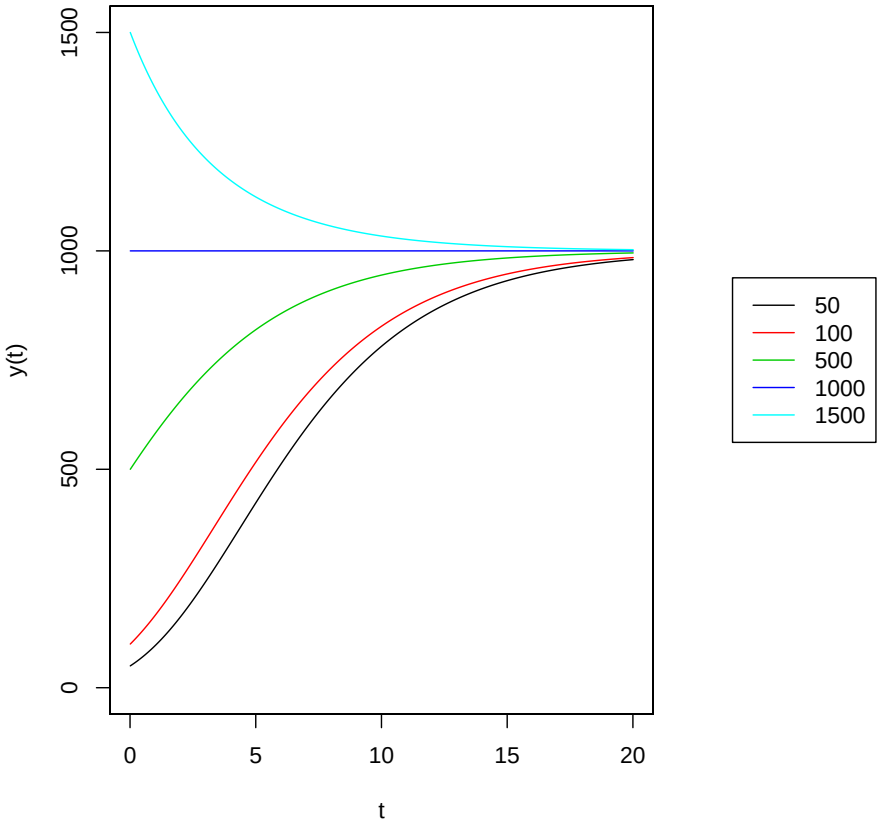


Figure 26.3: Graph produced using `ode_pop.R` set to solve $dy/dx = 0.25\ln(1000/y)$.

Chapter A: Getting Started with R

This is a very basic introduction to the elements of R that will be used in the very early part of this course. There is also extensive documentation on R available at `http://cran.r-project.org/doc/manuals/R-intro.html`.

R is a mathematical environment that allows you to easily solve many of the quantitative problems that arise in the life sciences. This document briefly describes some of the key elements in using R to

1. do descriptive statistics,
2. matrix algebra,
3. probability, and
4. discrete difference equations.

These are all topics that will be covered in detail in the course and this appendix is designed to get you started using R and describe some of the commonly used functions you will find in each of the "R Skills" sections near the end of each chapter of this text.

A.1 Starting R

To get started, open R. When the program opens, it should look like Figure A.1.

Observe the cursor blinking next to a `>` towards the bottom of the page. This is the command line in your R terminal. Type `getwd()`, then press your enter key. R will now display your working directory. Any files that you wish to upload or use, or eventually save, will be stored in this folder on your computer. To change your working directory at any time, use the `cd` command, followed by the path to your desired folder. To view files in your current directory, type `dir()`.

A.2 Working from the Command Window

Basic Arithmetic in R

Let us get started by trying out some basic arithmetic commands. We can do this right in the terminal, or Command Window as it will commonly be

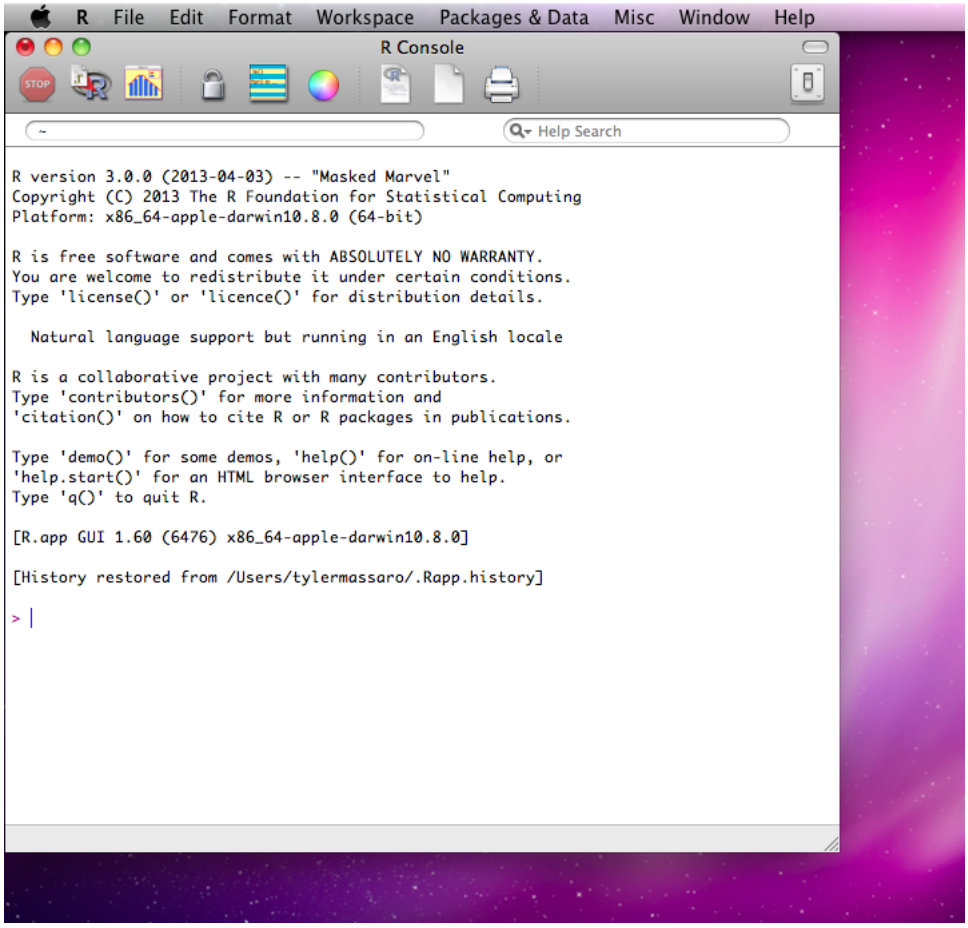


Figure A.1: What R looks like on opening.

referred to in this textbook. Try typing in the following commands. After typing each line, press the enter key.

```
> 5 + 7
> 6 - 9
> 2 * 9
> 25 / 2
> 3^5
```

R also has special commands for other common operations. Use the `log(x)` command to find the natural logarithm of the number x. For example, try typing into the Command Window

```
> log(5)
```

If you need to find the logarithm with a base other than $e \approx 2.71828183$, use the log property

$$log_a x = \frac{\ln x}{\ln a}.$$

For example, if you wanted to find $\log_2 5$, you would type

```
> log(5)/log(2)
```

into the Command Window.

R uses the command `exp(x)` to find e^x. For example, if you wanted to find $e^{5.46}$, you would type

```
> exp(5.46)
```

into the Command Window.

Some other commond commands are

- `sin(x)`, finds the sine of the number x,
- `cos(x)`, finds the cosine of the number x,
- `tan(x)`, finds the tangent of the number x,
- `factorial(x)`, finds $x! = x \cdot (x-1) \cdot (x-2) \cdots 2 \cdot 1$,
- `sqrt(x)`, finds the square root of the number x, i.e. $\sqrt{x}$.

Some Useful R Functions

Rounding

There are several different functions used for rounding depending on the rule you want to use for rounding. The function `round(x)` will round the number x to the nearest integer. It will round up if the decimal portion of x is greater than or equal to 0.5, and down otherwise. If you always want to round up, use the function `ceiling(x)` to round up to the nearest integer. If you always want to round down, use the function `floor(x)` to round down to the nearest integer. If you want to round to the 10^{-n}-th place, for some positive integer n, use `round(x,n)`.

Command Window

```
> pi
[1] 3.141593
> round(pi)
[1] 3
> ceiling(pi)
[1] 4
> floor(pi)
[1] 3
> round(pi, 2)
[1] 3.14
```

Finding Roots of Polynomials

A polynomial of degree n, written generally as

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

will have n roots (values of x where $P(x) = 0$), some of which may be imaginary numbers. To calculate the value of the roots of a polynomial in R, we use the function `polyroot(c)` where `c = c(`a_0`, `a_1`, ..., `a_n`)` is the array of coefficients of the polynomial in ascending order.

For example, suppose we want to find the roots of the polynomials

$$f(x) = x^2 - 4x - 5 \quad \text{and} \quad g(x) = \frac{1}{2} - 24x^2 + 64x^3 - 48x^4.$$

To find the roots of $f(x)$ we would use the command `polyroots(c(-5, -4, -1))`. Notice the coefficient in front of the x^2 term is 1. Additionally, we must make sure to include the negative signs in front of the coefficients of the x term and the constant term. To find the roots of $g(x)$ we would use the command `polyroot(c(0.5, 0, -24, 64, -48))`. Notice we list the coefficients in ascending order (starting with the lowest term first) even though the terms are not written in that order. Additionally, note that there is no x term in $g(x)$ so it has a coefficient of 0.

Command Window

```
> polyroots(c(-5, -4, 1))
[1] -1+0i  5+0i
> polyroot(c(0.5, 0, -24, 64, -48))
[1]  0.1928638-0.0000000i -0.1237411-0.0000000i
[3]  0.6321053-0.1921521i  0.6321053+0.1921521i
```

Thus, we find that for the function $f(x)$ the roots are $x = -1$ and $x = 5$. Indeed, if you evaluate $f(x)$ at $x = -1$ and $x = 5$, you will calculate a function value of 0. For the function $g(x)$ we find that there are two real roots and two imaginary roots (recall that imaginary roots always come in pairs).

A.3 Working with Arrays (Vectors and Matrices)

In this section we learn how to enter matrices into R and do some basic matrix computation.

Entering Vectors and Matrices into R

Suppose you have the following data

x	2	5	2	4	6
y	4	7	5	8	11

There are two different ways in which we can enter the data. The first method is to enter a vector for the x data and a vector for the y data. In the Command Window this would look like

Command Window

```
> x = c(2, 5, 2, 4, 6)
> x
[1] 2 5 2 4 6
> y = c(4, 7, 5, 8, 11)
> y
[1] 4 7 5 8 11
```

Notice that we use `c( , )` to construct a vector, with entries separated by commas. Also, notice that when you press the enter key, R will store the vector. To display it, simply type in the name you have assigned, and press the return key again.

Should you want to represent the x and y data as row vectors, instead of column vectors, use the `t()` command to take the transpose. This is slightly confusing, since R, by default, will automatically construct column vectors, but will display them as row vectors when called. In the code shown below, notice that, when displayed, the data now has the corresponding column index listed above the entry.

Command Window

```
> x = t(c(2, 5, 2, 4, 6))
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    5    2    4    6
```

To construct a matrix, we use the `matrix()` command. This is similar to our concatenate argument `c()` to construct vectors, but with the added option of defining the specific number of rows or columns we would like.

Command Window

```
> data = c(2, 5, 2, 4, 6, 4, 7, 5, 8, 11)
> M = matrix(data, ncol = 2)
> M
     [,1] [,2]
[1,]    2    4
[2,]    5    7
[3,]    2    5
[4,]    4    8
[5,]    6   11
```

Here, `ncol` allows us to set the number of columns we would like to use. We could have also specified `nrow = 5`.

You might want to check the length of a vector or the size of a matrix once you have entered it. Use the `length(x)` command to find the size of a vector labeled `x`. Use the `dim(A)` command to find the dimensions of a matrix labeled `A`. This function will output two numbers, the first being the number of rows in `A`, the second being the number of columns in `A`. Also, if you need the sum of all the elements in your vector or sum the entries of your matrix, use `sum(x)` or `sum(A)`, respectively, where `x` is a vector and `A` is a matrix.

Command Window

```
> length(x)
[1] 5
> dim(M)
[1] 5 2
> sum(x)
[1] 19
> sum(M)
[1] 54
```

Accessing Entries within a Matrix

Once we have our matrix of data, we would like to be able to access just the x data and just the y data if needed. R makes this easy. If we have a matrix `A`

entered into R, then we can use the structure `A[i,j]` to access portions of the matrix, where `i` indicates the row or rows that we want, and `j` indicates the column or columns we want. Use `A[1,1]` to get the entry in the first row, first column. The following sequence of commands entered into the Command Window will (1) check the entry in the second row, first column of the matrix `M`, (2) name the first column `x`, (3) name the second column `y`, (4) name the third row `foo`, and (5) create a new matrix called `newdata` that contains only rows 2 through 4.

Command Window

```
> M[2,1]
[1] 4
> x = M[,1]
> x
[1] 2 5 2 4 6
> y = M[,2]
> y
[1] 4 7 5 8 11
> foo = data[3,]
> foo
[1] 2 5
> newdata = M[2:4,]
> newdata
     [,1] [,2]
[1,]    5    7
[2,]    2    5
[3,]    4    8
```

Constructing Special Matrices in R

If you have an array of values that increase by regular intervals, say, a list of years, there is a shortcut to entering the data. Suppose you wanted to make a vector each year from 1980 to 2010. It would be tedious to enter each year by hand, so we use the short cut, `seq(a, b, by = c)` where `a` is the smallest value, `b` is the largest value, and `c` is the increment.

Command Window

```
> years = seq(1980, 2010, by = 1)
> years
 [1] 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989
[11] 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999
[21] 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009
[31] 2010
> every5years = seq(1980, 2010, by = 5)
```

```
> every5years
[1] 1980 1985 1990 1995 2000 2005 2010
```

Notice that when there are too many entries for R to print on one line, it will label which entry is leading the next row.

A.4 R scripts

So far, every command we have run we have typed into the Command Window. This is convenient for quick calculations, but often we have a series of commands we would like to run several times, perhaps with only a small modification each time. For this, R provides the capability of writing and running R scripts. An R script is a file that contains a series of commands, and can be run from the Command Window by typing `source("file.R")`.

Steps to Creating and Running an R script

1. Check the location of the current directory. Make sure the current directory is pointed to the folder where you want to save your R script. If necessary, change the location of the current directory.
2. Open a new R script (`File -> New Document`).
3. Type your commands into the R script. See an example of an R script below.
4. Save the R script in the current directory (`File -> Save As...`). Save the R script as *filename*.R where *filename* is the name of your file.
5. To run your R script, type `source("filename.R")` into the Command Window and press the enter key.

An example R script:

`test.R`

```
# This is a comment in an R script.  Anything typed after
# the pound sign in an R script will not be read as a
# command.  This allows you to put comments and notes in
# your R scripts.

x = seq(0, 10, by = 2.5)      # Enter some vector

m = length(x)                 # Find the length of x
```

If we save this file as `test.R` and run it in the Command Window using `source("test.R")`, nothing will show up on the screen afterwards. This is because, in the above code, we did not run any commands which would have printed information in the terminal. However, there is now an array `x`, whose length has been assigned to `m`. To verify this, simply type `x` or `m` into the Command Window, press the enter key, and observe what happens.

Notice that we can add lines into the file that are not interpreted as commands by putting a # at the front of the line or after a command.

A.5 Nicely Formatted Output: `cat(sprintf(...`

After you get the hang of writing R scripts, you will be able to write R scripts that generate a lot of different output. It is often useful to display this output in a nicely formatted way. For this we use the `sprintf` function, along with the `cat` function. These functions are best explained through some examples.

Suppose we have an array of numbers and we want to compute the minimum and maximum values in that array. We could use the following R script to do this.

MinMax.R

```
# Filename: MinMax.R
# R script to
#  - calculate the minimum & maximum of an array
#  - display t hat min and max

# Create array
x = c(20, 45, 81, 6, -3, -23, 99)

# Find minimum and maximum of array
minx = min(x)
maxx = max(x)

# Display the min & max using cat and sprintf
cat(sprintf("The max is %d\n", maxx))
cat(sprintf("The min is %d\n", minx))
cat(sprintf("The average of the min and max is %4.1f\n",
    (maxx+minx)/2))
```

Look at the first `cat(sprintf(...` line. The text we want displayed is in double quotes, `" "`, while the max value we want displayed is replaced with `%d`. The `%` indicates you want to insert a calculated value here, in this case, `maxx`. The `d` indicates it is a decimal value. The `\n` just before the close of the single quotes indicates that you want a new line. This way the next bit of information we print out will start on a new line. After the single quotes there

is a comma, followed by the name of the computed value we want inserted for `%d`. The second `cat(sprintf(...` line is similar to the first, only we display the minimum value.

Now, look at the last `cat(sprintf(...` line. Notice, we now use `%4.1f` instead of `%d`. The `f` indicates that this is a real number and we will not use scientific notation. If we did want to use scientific notation, we would use an `e`. The `4.1` indicates that we require four spaces to display our number and there will be one digit after the decimal. Note that the decimal point counts as a space. So `4.1` indicates that there will be two digits before the decimal, the decimal, and the one digit after the decimal, for a total of four spaces. The other difference to note in this line, is that we can do calculations within the `sprintf` command. The value we wish to display in the `%4.1f` slot, is computed using `maxx` and `minx`.